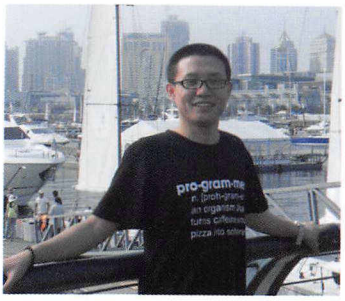
Broadview®
www.broadview.com.cn

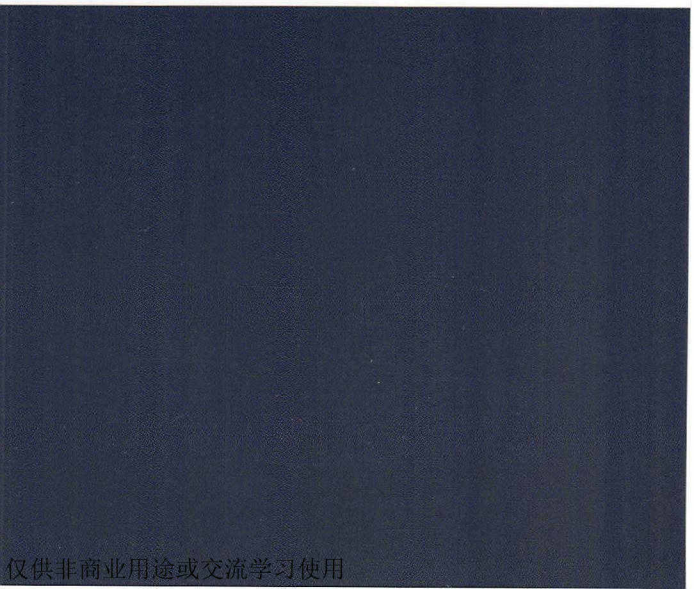
高级Android开发强化实战

王辰龙 编著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

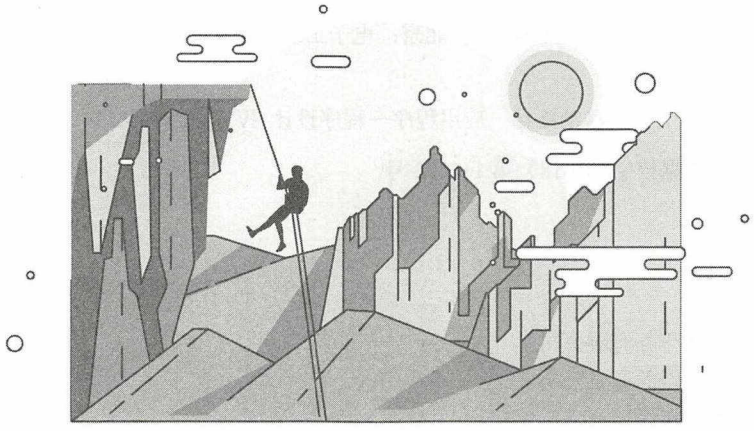


王辰龙，算法资深工程师，毕业于北京交通大学。在移动互联网公司从事多年技术研发工作，对于技术与产品有着深刻的理解。热爱开源，热爱分享，追求极致的技术实现。践行工匠精神，崇尚“艺无止境”。



高级Android开发强化实战

王辰龙 编著



電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书收集了约20个关于高级Android开发的进阶实例，这些实例都是对在日常开发中遇到的问题的抽象，涉及整个Android开发的各个技术栈。本书从常见的问题入手，引导读者逐步地掌握进阶的各个实例，同时提供分析和解决问题的思考过程，寻求最优方案。本书的内容分为以下几个部分。

进阶基础：通过管中窥豹，剖析Activity和View的一些基本概念，展示源码分析的常见方法；高阶控件：讲解MD的两个复合布局和约束布局，介绍高级控件的开发流程；项目架构：架构是项目的骨骼，该部分介绍主流的MVP系列和Flux架构；响应式编程：解析响应式编程三剑客RxJava+Dagger+Retrofit的不同特性组合使用方法；功能与动画：列举若干实际开发中的经典实例，包含功能定制和页面动画等；Kotlin与SVG：讲解Kotlin编程语言和SVG图像技术的若干开发技巧；测试与优化：介绍自动化测试框架的设计方法，以及优化应用的常用工具。

通过对本书的学习，读者可以极大地提高Android开发的工程能力，从而成为一名合格的高级Android工程师，不仅在理论上有所提升，在实践中也能直接应用。高级Android工程师通过对本书的学习也能完善知识体系和技术栈。

图书在版编目（CIP）数据

高级 Android 开发强化实战 / 王辰龙编著. —北京：电子工业出版社，2018.7
ISBN 978-7-121-34298-1

Ⅰ. ①高… Ⅱ. ①王… Ⅲ. ①移动终端－应用程序－程序设计 Ⅳ. ①TN929.53

中国版本图书馆 CIP 数据核字（2018）第 111022 号

策划编辑：张国霞
责任编辑：宋亚东
印　　刷：三河市兴达印务有限公司
装　　订：三河市兴达印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　印张：18.5　字数：413 千字
版　　次：2018 年 7 月第 1 版
印　　次：2018 年 7 月第 1 次印刷
定　　价：89.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

仅供非商业用途或交流学习使用

前　言

在编程之余，有时候我就在想，什么样的程序员属于高级程序员呢？或者说，高级程序员有哪些特性呢？工作年限一定不是一个关键的指标，许多工作多年的程序员依然写不出优雅的程序。无论是在 Android 开发还是其他领域，高级程序员一定是勤奋的，可以快速地掌握大量的新技术、新框架，不仅懂得原理，还能把新的技术落地到公司的产品中去。这是衡量程序员工作能力的一个重要标准，那么怎样才能将技术运用自如呢？唯有实践。基于此，我想把自己在日常实践中的一些经典案例，编著成一本成体系的书，以便为想要进步的 Android 程序员增加更多的实战经验，这也是编写本书的核心目的所在。

编写本书的另外一个目的，是帮助程序员建立产品的思想，对于技术而言，孤立的存在是没有任何意义的，技术只有与需求相结合，才能具有自身的价值。技术人员在开发的过程中，要时刻了解所完成的功能可以为公司带来哪些价值，是提升用户的访问兴趣，还是提升用户的使用流畅度，抑或是其他。当以产品思维去思考技术的时候，就会有动力、有目的地学习更多有价值的技术，而不是哗众取宠地学一些“看似有用”的新技术。

除此之外，还有理解架构的本质。一些技术人员经常会问：“为什么要使用架构？这只会增加额外的代码量，而且并不会对功能或性能有所提升，只需要完成必要的开发任务即可。”这种想法是非常浅显的，因为任何一个应用都不是一次成型的，需要不断地迭代，不断地扩展，同时需要不断地修改已有的业务逻辑，这就会涉及系统兼容性的问题。如何修改新的业务逻辑而不影响旧的业务逻辑？如何最大限度地复用已有的业务逻辑？架构就是解决这类问题的钥匙，一个优秀且合适的项目架构可以保证系统的稳定性，当开发新的功能或者修改旧的功能时，不至于破坏已有的业务体系。

本书的实例都是经典实战实例，每一个例子都代表一类在开发中需要掌握的技巧。本书由浅入深地逐个讲解需要掌握的开发理论与实践，共分为七部分。

◎ 进阶基础：偏重于源码的解析和理解，介绍阅读源码的技巧，以 Activity 和 View 为例，管中窥豹地分析其中的基础知识。读者也可依据此类方法，分析其他系统

组件的源码。

- ◎ 高阶控件：介绍若干复杂的控件布局，即 AppBarLayout、CoordinatorLayout 和 ConstraintLayout。通过实例，让读者理解在复杂控件布局中子视图是如何组合和相互关联的。
- ◎ 项目架构：分析主流架构的设计思想，即 Google 推荐的 MVP 和 MVVM，还有 Facebook 的 Flux。理解这些架构是如何组织和管理大型项目的，以及它们的优点和缺点各有哪些。
- ◎ 响应式编程：响应式是一种编程思想，在处理网络请求和功能测试时，基于响应式框架的项目拥有更好的可扩展性和可维护性，响应式编程三剑客即 RxJava、Dagger 和 Retrofit。
- ◎ 酷炫功能与精美动画：实现两个稍复杂的功能，分别是基于第三方控件和基于系统控件的扩展；并实现两个动画效果，分别用于页面展开和页面切换。对于功能和动画，不同的需求或样式层出不穷，本部分侧重于开发思路的分享。
- ◎ Kotlin 与 SVG：Kotlin 是用于替代 Java 的高阶编程语言，SVG 是用于替代 PNG 格式的图像技术，本书着重介绍一些基础概念，提纲挈领，以便于读者后续进行自主学习。
- ◎ 测试与优化：分享一个主流的自动化测试框架，以及优化内存与电量的一些常见方法。产品的性能与功能同样重要，应用的高品质也会提升产品的用户体验。

这七部分几乎已经包含高级Android编程的全部内容，本书的每个部分都会通过多个实例，从不同的角度引领读者亲身实战，真正地掌握高级编程的核心开发技巧。但是，实例的数量终究有限，希望读者更多地关注于实战中的开发思想，而不是具体的代码逻辑，代码总会不断地更迭，解决问题的思维却历久弥新。本书中的实例更多的是以点带面，读者可以一边阅读和思考，一边编写代码，相信读完本书，一定受益匪浅；同时，通过本书的实例可以解决一些常见的开发需求。衷心希望每位读者在阅读完本书之后，都“不虚此行”！

将本书送给我正在怀孕的妻子，感谢你在生活和工作中给予我的支持和帮助。

王辰龙

2018 年 5 月于北京海淀

目　　录

第 1 章
进阶基础

1.1 深入剖析 Activity 的生命周期

在 Android 系统中，所有应用都是由系统提供的四大组件构成的，即 Activity（活动）、Service（服务）、ContentProvider（内容提供者）和 BroadcastReceiver（广播接收器），而它们之间通过 Intent（意图）进行通信并相互关联。其中，Activity 作为与用户打交道最为频繁的组件，承担起显示的重任，其重要性不言而喻。因此，在开发中，对于 Activity 的全部工作细节，需要熟练掌握。

在 Android 系统中使用生命周期（Lifecycle）描述组件在不同状态之间的切换。通过管理组件的生命周期实现资源的获取与释放，在不同阶段的生命周期中触发相应的回调函数，在回调函数的上层方法中，系统控制组件的创建与销毁。同样，Activity 作为组件之一，也有着一套有序的生命周期，管理 Activity 的全部行为。

图 1-1 展示的是 Activity 的完整生命周期（图片来源于 Android 官网）。

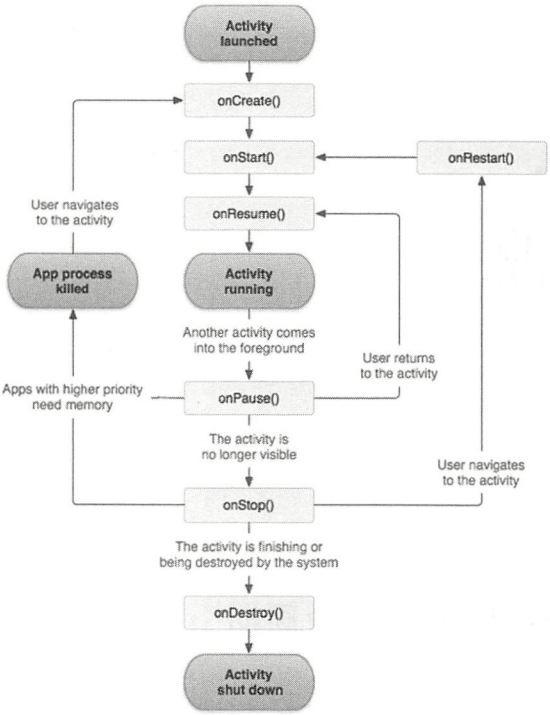


图 1-1　Activity 的完整生命周期

在系统中，有两种情况可以触发生命周期的改变，第 1 种是有用户参与的生命周期的改变，第 2 种是因为系统回收或配置修改而导致的生命周期的改变。为了更加直观地展示这两种改变生命周期的行为，本书编写了一个相关实例。通过这个实例，可以更加清晰地理解 Activity 的生命周期的变换形态。实例分为三个部分，将有用户参与的生命周期的变化分为因硬件导致的生命周期变化、在页面切换时生命周期的变化及由系统原因导致的生命周期的变化。

该实例的完整代码的下载地址为 https://github.com/SpikeKing/wcl-activity-lifecycle-demo。

1.1.1　Activity 的生命周期的各种状态

在 Activity 中，生命周期状态主要包含 6 种，即 onCreate()、onStart()、onResume()、onPause()、onStop()和 onDestroy()。每种生命周期都代表着 Activity 处于不同的状态：onCreate()表示 Activity 的创建，onStart()表示 Activity 的启动，onResume()表示 Activity 的恢复，onPause()表示 Activity 的暂停，onStop()表示 Activity 的停止，onDestroy()表示 Activity 的销毁。

```
// Activity组件
public class Activity extends ApplicationContext {
```

```
    // Activity 的创建
    protected void onCreate(Bundle savedInstanceState);

    // Activity 的启动
    protected void onStart();

    // Activity 的恢复
    protected void onResume();

    // Activity 的暂停
    protected void onPause();

    // Activity 的停止
    protected void onStop();

    // Activity 的销毁
    protected void onDestroy();
}
```

这些状态之间两两配对、有始有终，构成三组生命周期，相互嵌套。这三组生命周期分别为完整的生命周期（Entire Lifetime）、可视的生命周期（Visible Lifetime）及前台的生命周期（Foreground Lifetime）。

1）完整的生命周期：表示 Activity 组件从创建到销毁的全部过程，是最外层的生命周期。生命周期发生在调用 onCreate()方法与调用 onDestory()方法之间。在 onCreate()的方法中，系统会创建 Activity 的全局状态，例如填充布局等；在 onDestroy()方法中，系统会释放 Activity 所保存的资源。需要注意的是，onDestroy()不能保证被调用的时机，例如 Activity 在 Activity 栈中，在系统内存不足时，就可能触发强制销毁调用 onDestroy()方法，而在其他情况下，只有当 Activity 出栈时，才会调用 onDestroy()方法。

2）可视的生命周期：表示 Activity 组件从用户可视到离开用户视线的全过程。在这期间，用户可以在屏幕上看见当前的 Activity。生命周期发生在调用 onStart()方法与 onStop()方法之间。在 onStart()方法和 onStop()方法中，执行与用户可视相关的逻辑。因为 onDestroy()方法并不一定会被调用，所以在 onStop()方法中，一般会执行保持当前 Activity 状态的逻辑。在 Activity 的全部存活时间中，onStart()方法和 onStop()方法可能会被多次调用，这是因为 Activity 可能交替地由可视状态到不可视状态，再恢复到可视状态。

3）前台的生命周期：表示 Activity 组件显示于其他 Activity 组件之前，即位于 Activity 任务栈的栈顶，拥有最高优先级的资源使用权限，同时可以获得用户的输入焦点与用户进行交互。在可视的生命周期中，Activity 组件可能位于全透明或部分透明的 Activity 下面，即属于可视状态。而前台状态必须位于全部的 Activity 之上。生命周期发生在调用 onResume()方法与 onPause()方法之间。在 onResume()方法和 onPause()方法中，执行与交互相关的逻辑。同样，在 Activity

的全部存活时间中，onResume()和 onPause()也可能会被多次调用，即 Activity 可能会频繁地获取与失去焦点。

这三类生命周期循环嵌套，完整的生命周期包含可视的生命周期，可视的生命周期又包含前台的生命周期，它们共同构成 Activity 的生命周期体系。除此之外，还有三个特殊的生命周期状态，即 onRestart()、onRestoreInstanceState()、onSaveInstanceState()。当页面从 Activity 栈内调至栈顶时，会调用 onRestart()方法，初次创建时不会调用；onSaveInstanceState()用于储存 Activity 的状态信息；onRestoreInstanceState()用于恢复 Activity 的状态信息，仅用于系统导致的页面重建，而用户导致的页面重建需要在 onCreate()中由开发者自主恢复状态信息。

1.1.2 实例：准备

下面通过实例，让我们更好地理解生命周期的状态切换。实例分为三个部分：因硬件导致的生命周期变化、在页面切换时导致的生命周期变化、由系统原因导致的生命周期变化。在讲解实例之前，先展示本实例的运行环境，以便读者更好地理解与执行代码，本书其他实例的运行环境与此实例类似。动手实践是学习编程的不二法门，也是成为高级程序员的必要条件之一。

1．硬件环境

实验真机为 Redmi Note 4X，即红米 Note 4X；操作系统为 Android 6.0。

实验模拟器：操作系统为 Android 7.1.1，如图 1-2 所示。

图 1-2 模拟器（开发电脑：MacBook Pro 10.11.6）

IDE（Integrated Development Environment）：Android Studio 2.3.2，如图 1-3 所示。

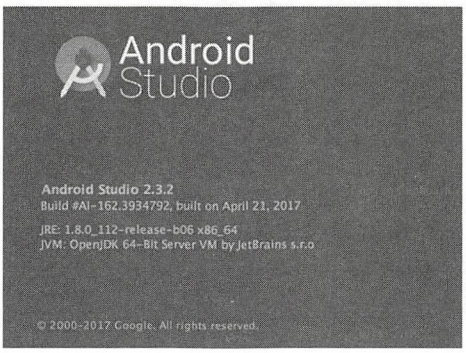


图 1-3　IDE

2．软件环境

Android Tools 的 Gradle 版本为 2.2.3，位于根目录下的 build.grale 文件中

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:2.2.3'
// Android Tools 的 Gradle 版本

        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}
```

Android 的 Gradle 构建版本为 2.14.1，位于根目录下的

```
/gradle/wrapper/gradle -wrapper.properties 中:
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
distributionUrl=https\://services.gradle.org/distributions/gradle-2.14.1-all
.zip
```

Android SDK 版本为 24，位于项目目录下的/app/build.gradle 中：

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 24
```

```
        buildToolsVersion '24.0.0'

        defaultConfig {
            applicationId "org.wangchenlong.xxx"
            minSdkVersion 14
            targetSdkVersion 24
            versionCode 1
            versionName "1.0"
        }
        buildTypes {
            release {
                minifyEnabled false
                proguardFiles       getDefaultProguardFile('proguard-android.txt'),
'proguard-rules.pro'
            }
        }
    }

    dependencies {
        compile fileTree(dir: 'libs', include: ['*.jar'])
        testCompile 'junit:junit:4.12'
        compile 'com.android.support:appcompat-v7:24.0.0'
    }
```

全部实例代码都已经通过测试，从网络下载后，运行项目即可使用。读者在阅读本节时，需要提前下载源码，并调试通过，根据所讲述的内容运行代码，通过实战更好地学习本节知识。

1.1.3 实例：因硬件导致的生命周期变化

我们在使用应用的过程中，除在应用内的交互外，通过手机的硬件按键，也能触发生命周期的改变。对于应用内的交互会导致页面的切换，则触发常规的生命周期状态的改变，从 onCreate()到 onDestroy()依次调用，这是基本的生命周期变化。除此之外，还需重点关注一些特定的硬件操作，比如用户按手机的菜单（Home）键和后退（Back）键。

对于一款基本的 Android 手机而言，手机底部都会包含两个基础的按钮，一个是位于中部的菜单键，另一个是位于右侧的后退键。因此，除了在应用内提供的交互场景，用户最常用的两种操作就是按菜单键或后退键。那么这两种常用的操作又会给 Activity 带来哪些生命周期的变化呢？下面通过实例来讲解。

先来看看应用的主页 - MainActivity.java，通过源码来分析这个 Activity 页面是如何组成的，如图 1-4 所示。

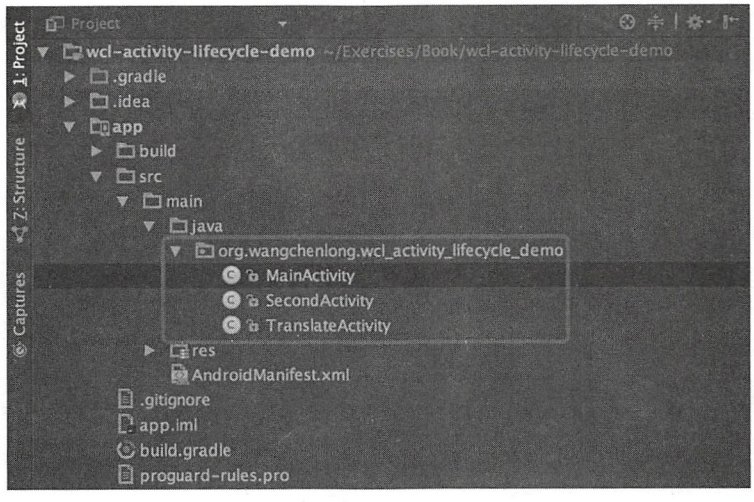


图 1-4　MainActivity

首先，声明 TAG 标签，作为 Log 输出的标签，用于在 Logcat 中过滤信息；使用 ButterKnife 插件，声明两个按钮，用于页面的跳转；声明 EXTRA_TEXT 标签，用于在 Activity 中模拟状态的存储。对于字符串静态常量，一般使用大写字母加下画线的形式表示。

代码如下：

```
    // Log 输出的标签，用于在 logcat 中过滤
    private static final String TAG = "DEBUG-WCL: " + MainActivity.class.
getSimpleName();
```

```
    @Bind(R.id.main_b_translucent)
    Button mBTranslucent; // 透明页面
    @Bind(R.id.main_b_second)
    Button mBSecond; // 第二个页面

    private static final String EXTRA_TEXT = "extra_text";
    // 用于在 Activity 中模拟状态的存储
```

注意：ButterKnife 是由 Jake Wharton 编写的一个用于声明布局中控件的插件，代替调用 findViewById()方法，节省了大量的开发时间。在项目目录中，添加 ButterKnife 的依赖即可使用。具体原理与使用方式可参考 ButterKnife 的官方网址 https://github.com/JakeWharton/butterknife。

其次，onCreate()方法作为创建 Activity 的第一个调用函数，负责初始化 Activity 的全部行为，我们一般重点编写代码逻辑的部分。例如 setContentView()，设置页面布局；ButterKnife.bind()，负责绑定 ButterKnife 并初始化，只有在初始化 ButterKnife 之后，才能调用 Bind()标注进行布局

ID 的绑定；设置 mBTranslucent 按钮和 mBSecond 按钮的绑定事件；获取 EXTRA_TEXT 的状态存储信息及输出 onCreate()方法的调用提示。在每个生命周期方法中，都添加了相应的调用提示，方便在 Logcat 中观察当 Activity 页面改变时，生命周期变化的整个过程。

代码如下：

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        ButterKnife.bind(this); // 绑定 ButterKnife，并初始化

        mBTranslucent.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                startActivity(new    Intent(MainActivity.this,    TranslateActivity.
class));
            }
        });

        mBSecond.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                startActivity(new Intent(MainActivity.this, SecondActivity.class));
            }
        });

        // 获取 EXTRA_TEXT 的状态存储信息
        if (savedInstanceState != null) {
            String txt = savedInstanceState.getString(EXTRA_TEXT);
            Log.e(TAG, "[onCreate]savedInstanceState: " + txt);
        }

        Log.e(TAG, "onCreate"); // 调用提示
    }
```

再次，在 Activity 的启动过程中，先调用 onCreate()方法，然后依次调用 onRestart()、onStart()、onRestoreInstanceState()、onResume()，其中 onRestart()与 onRestoreInstanceState()并不是每个生命周期都会调用的。

代码如下：

```
    @Override
    protected void onRestart() {
        super.onRestart();

        Log.e(TAG, "onRestart");
```

```
}

@Override
protected void onStart() {
    super.onStart();

    Log.e(TAG, "onStart");
}

@Override
protected void onRestoreInstanceState(Bundle savedInstanceState) {
    super.onRestoreInstanceState(savedInstanceState);

    // 恢复状态
    if (savedInstanceState != null) {
        String txt = savedInstanceState.getString(EXTRA_TEXT);
        Log.e(TAG, "[onRestoreInstanceState]savedInstanceState: " + txt);
    }

    Log.e(TAG, "onRestoreInstanceState");
}
@Override
protected void onResume() {
    super.onResume();

    Log.e(TAG, "onResume");
}
```

最后，在 Activity 的关闭过程中，依次调用 onPause()、onSaveInstanceState()、onStop()、onDestroy()，其中 onSaveInstanceState()并不是每个生命周期都会调用的，只有页面完全不可视时才会调用，用于储存当前页面的状态，如果仅仅失去焦点则不调用。

代码如下：

```
@Override
protected void onPause() {
    super.onPause();

    Log.e(TAG, "onPause");
}

@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);

    // 写入状态
    if (outState != null) {
        outState.putString(EXTRA_TEXT, "C.L.Wang");
    }
```

```
    Log.e(TAG, "onSaveInstanceState");
}

@Override
protected void onStop() {
    super.onStop();

    Log.e(TAG, "onStop");
}

@Override
protected void onDestroy() {
    super.onDestroy();

    Log.e(TAG, "onDestroy");
}
```

最终 MainActivity 页面的显示如图 1-5 所示。

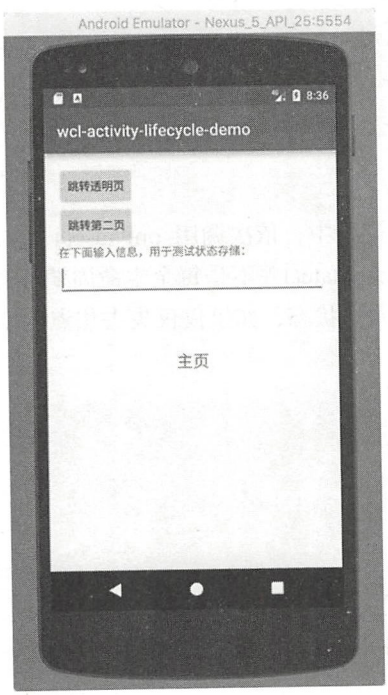


图 1-5　MainActivity 页面

在页面初次启动时，MainActivity 页面执行的生命周期依次为 onCreate()、onStart()、onResume()，都是常规的生命周期流程。在执行 onResume()方法之后，MainActivity 页面已经具备与用户交互的功能。

```
MainActivity: onCreate
MainActivity: onStart
MainActivity: onResume
```

在按后退键时，MainActivity 页面执行的生命周期依次为 onPause()、onStop()、onDestroy()，也都是常规的生命周期流程。在执行 onDestroy()方法之后，MainActivity 页面已经完全销毁。

```
MainActivity: onPause
MainActivity: onStop
MainActivity: onDestroy
```

但是，如果没有按后退键，而是按菜单键，MainActivity 页面仅仅是由前台转入后台，并不会立即执行销毁。当用户再次使用时页面仍可快速启动。但是系统可能会在某个时刻自动销毁这个页面，如内存不足，这是 Android 系统的机制，防止后台页面过多，影响系统效率。

在按菜单键时，MainActivity 页面执行的生命周期依次为 onPause()、onSaveInstanceState()、onStop()，不会执行 onDestroy()方法，也就没有释放系统资源，保证在下次启动时，仍可快速加载。调用 onSaveInstanceState()方法的原因是系统可能在后台自动杀死当前 Activity，所以需要保存当前页面的状态，如果系统这样做，再次调起页面时，系统也会自动调用 onRestoreInstanceState()恢复当前状态。在执行 onStop()方法之后，MainActivity 页面已经由前台转入后台，对用户不可视。

```
MainActivity: onPause
MainActivity: onSaveInstanceState
MainActivity: onStop
```

在系统桌面中再次点击应用图标，应用又恢复为可交互状态，MainActivity 页面执行的生命周期依次为 onRestart()、onStart()、onResume()。由于间隔时间较短，系统资源较为丰富，系统未执行 onDestroy()方法，强制杀死当前 Activity，所以不需要调用 onCreate()方法进行页面创建，也不需要调用 onRestoreInstanceState()方法恢复状态。

```
MainActivity: onRestart
MainActivity: onStart
MainActivity: onResume
```

1.1.4　实例：页面切换时的生命周期变化

对于非透明页面，Activity 的逻辑比较固定，执行完整的生命周期，但是对于透明页面略有不同，因为透明页面作为底部页面的浮层，底部页面仍然可视，所以不会执行销毁逻辑。为了验证全部的生命周期过程，下面通过实例，创建非透明和透明两个页面，验证在 Activity 页面

切换时生命周期的变化情况。

基本的生命周期逻辑：在 onStart()时，Activity 页面对于用户可视，但是无法交互；在 onResume()时，Activity 页面不仅可视，而且可以交互，因此 onResume()在 onStart()之后触发；在 onPause()时，Activity 页面无法交互，而且必须在 Activity 页面的 onPause()执行完成后，下一个 Activity 页面才能执行启动逻辑，如 onCreate()或 onResume()；在 onStop()时，Activity 页面对于用户不可视。

注意：在 onPause()中，不能执行复杂的操作，否则会影响下一个 Activity 的启动速度。

从非透明页面讲，首先创建一个简单的非透明页面，此页面的创建生命周期仅含有 onCreate()，销毁生命周期全部包含，即 onPause()、onStop()、onDestroy()。

```
public class SecondActivity extends AppCompatActivity {
    private static final String TAG = "DEBUG-WCL: " + SecondActivity.class.
getSimpleName();

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_second);
        Log.e(TAG, "onCreate"); // 生命周期
    }

    @Override
    protected void onPause() {
        super.onPause();
        Log.e(TAG, "onPause"); // 生命周期
    }

    @Override
    protected void onStop() {
        super.onStop();
        Log.e(TAG, "onStop"); // 生命周期
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        Log.e(TAG, "onDestroy"); // 生命周期
    }
}
```

首先，执行在 MainActivity 中启动非透明页面 SecondActivity，即由当前 Activity 页面切换至另一个非透明的 Activity 页面，导致当前页面销毁，新的页面被创建。在 MainActivity 页面

的 onPause()执行完成后，SecondActivity 页面才能执行启动逻辑，SecondActivity 启动完成后，才会执行 MainActivity 页面的剩余销毁工作，如 onSaveInstanceState()和 onStop()。因为当前 Activity 页面仍在 Activity 栈中，并未出栈，则并未执行 onDestroy()的方法。

```
MainActivity: onPause
SecondActivity: onCreate
MainActivity: onSaveInstanceState
MainActivity: onStop
```

注意：对于 onSaveInstanceState()方法，在页面关闭时都会执行，但是非系统原因的关闭，不会执行默认的恢复数据操作，即 onRestoreInstanceState()方法。

接着，关闭 SecondActivity 页面，重新返回 MainActivity 页面。因为 MainActivity 页面并未被销毁，即未执行 onDestroy()方法，所以无须重建，执行 onCreate()方法，只需要执行 onRestart()、onStart()、onResume()方法即可。同样，在 SecondActivity 执行 onPause()方法之后，MainActivity 才开始自身的创建工作。因为 SecondActivity 位于 Activity 栈的栈顶，所以触发出栈操作，页面完全销毁，所以执行 onStop()完成之后，会继续执行 onDestroy()方法。

```
SecondActivity: onPause
MainActivity: onRestart
MainActivity: onStart
MainActivity: onResume
SecondActivity: onStop
SecondActivity: onDestroy
```

启动一个透明页面。透明页面的特点是在启动之后，底部页面仍然可视，因此系统执行 Activity 生命周期的逻辑也有所变化。为了保证生命周期的逻辑清晰，透明页面与非透明页面的逻辑相同，生命周期状态仅显示 onCreate()、onPause()、onStop()、onDestroy()等。

```
public class TranslateActivity extends AppCompatActivity {
    private static final String TAG = "DEBUG-WCL: " + TranslateActivity.
class.getSimpleName();

    @Override
    public void onCreate(Bundle savedInstanceState, PersistableBundle
persistentState) {
        super.onCreate(savedInstanceState, persistentState);
        Log.e(TAG, "onCreate"); // 生命周期
    }

    @Override
    protected void onPause() {
        super.onPause();
        Log.e(TAG, "onPause"); // 生命周期
    }
```

```
    @Override
    protected void onStop() {
        super.onStop();
        Log.e(TAG, "onStop"); // 生命周期
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        Log.e(TAG, "onDestroy"); // 生命周期
    }
}
```

唯一不同的是，在透明 Activity 的声明 AndroidManifest.xml 文件中，使用透明主题样式。

```
<activity
    android:name=".TranslateActivity"
    android:theme="@style/AppThemeTranslucent"/>
```

透明主题样式如下：

```
<!--透明主题样式-->
<style name="AppThemeTranslucent" parent="Theme.AppCompat.Light.NoActionBar">
    <item name="android:windowNoTitle">true</item>
    <item name="android:colorBackgroundCacheHint">@null</item>
    <item name="android:windowIsTranslucent">true</item>
    <item name="android:windowAnimationStyle">@android:style/Animation</item>
    <item name="android:windowBackground">@color/colorTranslucent</item>
</style>

<!--透明颜色淡黑色-->
<color name="colorTranslucent">#88888888</color>
```

最终透明的 TranslateActivity 页面的显示如图 1-6 所示，按后退键即可关闭。

首先，在 MainActivity 中，启动透明页面 TranslateActivity。在启动透明页面时，MainActivity 页面仅仅执行 onPause()方法，与非透明页面不同，并不会继续调用 onStop()方法。因为 onPause()方法控制 MainActivity 是否具有交互功能，onStop()方法控制 MainActivity 是否可视，所以 MainActivity 透明页面 TranslateActivity，仅仅使其自身失去交互功能，并未影响其是否可视。

```
MainActivity: onPause
```

接着，关闭透明页面 TranslateActivity，重新恢复 MainActivity 页面。在 TranslateActivity 页面执行 onPause()之后，MainActivity 开始重建工作，因为启动 TranslateActivity 页面时，MainActivity 仅仅失去交互功能，所以只调用 onResume()恢复交互功能，并未执行 onStart()方法恢复可视。在 MainActivity 页面创建完成后，TranslateActivity 执行自身的销毁流程，即 onStop()、onDestroy()。

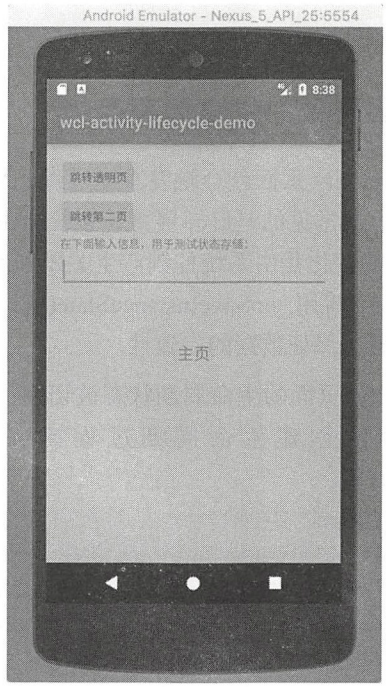


图 1-6　最终透明的 TranslateActivity 页面的显示

```
TranslateActivity: onPause
MainActivity: onResume
TranslateActivity: onStop
TranslateActivity: onDestroy
```

此处的执行操作顺序是：启动透明页、关闭透明页、启动非透明页、关闭非透明页。观察 Activity 在这一连续操作中生命周期变化情况。

```
MainActivity: onPause
MainActivity: onSaveInstanceState
TranslateActivity: onPause
MainActivity: onResume
TranslateActivity: onStop
TranslateActivity: onDestroy
MainActivity: onPause
SecondActivity: onCreate
MainActivity: onSaveInstanceState
MainActivity: onStop
SecondActivity: onPause
MainActivity: onRestart
MainActivity: onStart
MainActivity: onResume
SecondActivity: onStop
```

```
SecondActivity: onDestroy
```

1.1.5 实例：由系统原因导致的生命周期变化

当系统配置发生改变时，Activity 页面就会触发重建过程，因为系统配置会导致页面的某些参数发生变化，显示样式不同。最常见的是旋转屏幕，系统会由竖屏转为横屏，启用新的资源文件组。因为此次 Activity 页面的变化是由系统控制产生关闭和启动 Activity 的行为触发生命周期，所以 Activity 页面不仅会调用 onSaveInstanceState()触发保存数据，还会同步执行 onRestoreInstanceState()恢复数据，保证数据的一致性。

当旋转手机时，MainActivity 页面的生命周期状态被切换，先是执行页面的销毁生命周期过程流，然后是执行页面的创建生命周期过程流，注意 onSaveInstanceState() 和 onRestoreInstanceState()的调用时机。

```
MainActivity: onPause
MainActivity: onSaveInstanceState
MainActivity: onStop
MainActivity: onDestroy
MainActivity: onCreate
MainActivity: onStart
MainActivity: onRestoreInstanceState
MainActivity: onResume
```

最终，旋转之后页面的显示如图 1-7 所示。

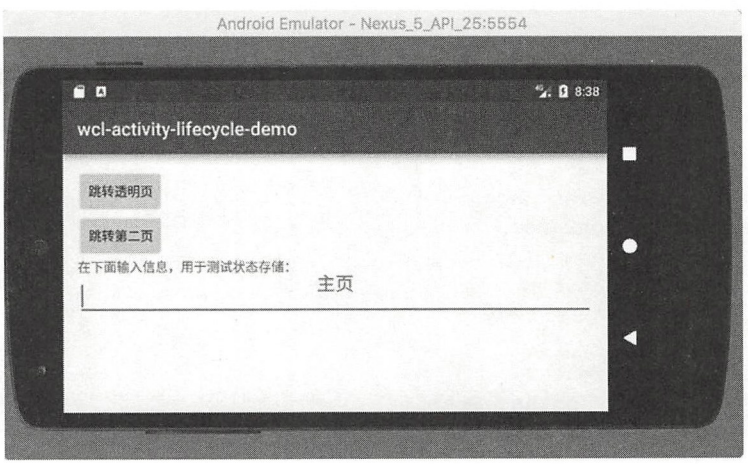


图 1-7 旋转之后页面的显示

注意：onRestoreInstanceState()方法需要在可交互方法（即 onResume()方法）之前执行，确保数据的正确恢复，显示无误；同时，onSaveInstanceState()方法需要在停止交互（即 onPause()方法）之后执行，确保用户不进行额外的输入，储存全部信息。

同时，系统也会保存 Activity 页面关闭之前的视图结构和暂存数据。在一些系统内置的视图中，如 TextView，具体保存哪些数据项，可以通过阅读 Android 的官方文档，也可以直接阅读 Android 源码获取。对于源码位置，搜索某个视图类源码中的 onSaveInstanceState()方法就可以找到。其中的 SavedState 类，就是保存数据的具体类，观察对于 SavedState 的具体操作，就知道在关闭 Activity 时，系统会强制保存哪些状态。

下面以 TextView 为例，观察在 onSaveInstanceState()中都保存了哪些数据。

```
@Override
public Parcelable onSaveInstanceState() {
    Parcelable superState = super.onSaveInstanceState();

    // Save state if we are forced to
    boolean save = mFreezesText;
    int start = 0;
    int end = 0;

    if (mText != null) {
        start = getSelectionStart();
        end = getSelectionEnd();
        if (start >= 0 || end >= 0) {
            // Or save state if there is a selection
            save = true;
        }
    }

    if (save) {
        SavedState ss = new SavedState(superState);
        // XXX Should also save the current scroll position!
        ss.selStart = start;
        ss.selEnd = end;
        if (mText instanceof Spanned) {
            Spannable sp = new SpannableStringBuilder(mText);

            if (mEditor != null) {
                removeMisspelledSpans(sp);
                sp.removeSpan(mEditor.mSuggestionRangeSpan);
            }

            ss.text = sp;
        } else {
```

```
            ss.text = mText.toString();
        }

        if (isFocused() && start >= 0 && end >= 0) {
            ss.frozenWithFocus = true;
        }

        ss.error = getError();

        if (mEditor != null) {
            ss.editorState = mEditor.saveInstanceState();
        }
        return ss;
    }

    return superState;
}
```

TextView 的核心状态存储类 SavedState，由 ss.selStart 和 ss.selEnd 保存光标（Selection）的起始位置与结束位置，即文本中的选中状态，在多数手机上，状态呈连续蓝色选中条；由 ss.text 保存内容（Text）信息；由 ss.frozenWithFocus 保存焦点信息；由 ss.error 保存错误信息；由 ss.editorState 保存可编辑信息，因为 EditText 继承于 TextView，所以在 TextView 中储存可编辑信息。其他系统内置的 View 与 TextView 类似，都是在 onSaveInstanceState()方法中保存 Activity 的状态。

如果 Activity 需要恢复状态数据，在恢复时机的选择上，可以选择 onCreate()或 onRestoreInstanceState()。区别是：在 onCreate()中，需要判断传入参数 savedInstanceState 是否为空，在某些情况下未储存数据；而在 onRestoreInstanceState()中，参数 savedInstanceState 确定非空，则不会调用这个方法。对于由系统原因导致的 Activity 页面变化，触发的生命周期改变，推荐使用 onRestoreInstanceState()恢复数据，针对性更强。因为 onCreate()在每次 Activity 重建时，都会被调用，无法区分是否因系统原因导致。

在 onCreate()中，恢复状态参数 savedInstanceState 的代码逻辑如下，即先判空，再操作：

```
if (savedInstanceState != null) {
    String txt = savedInstanceState.getString(EXTRA_TEXT);
    Log.e(TAG, "[onCreate]savedInstanceState: " + txt);
}
```

对于 Activity 页面是否支持屏幕旋转，系统默认支持这项操作。也可以通过改变在 AndroidManifest 中增加对于 Activity 的配置，设置不同的屏幕旋转状态模式实现。

比如，在 AndroidManifest 中的 Activity 属性中添加如下信息：

```
android:configChanges="orientation|screenSize"
```

这样就会防止在旋转屏幕时系统重绘 Activity 页面，但是手机旋转仍可触发 Activity 的某些监听。此时不会调用生命周期状态的方法，转而调用 onConfigurationChanged()方法，供开发者处理旋转屏幕事件。

再如，在 AndroidManifest 中的 Activity 属性中添加如下信息：

```
android:screenOrientation="portrait"
```

就会使屏幕保持竖直状态，Activity 页面无法旋转，也无法监听旋转状态。这样使用户体验始终保持一致，即屏幕竖直，也避免系统在旋转屏幕时强制重构 Activity，影响用户体验。

本章通过分析 Activity 页面的全部生命周期状态方法，全面梳理了 Android 系统处理 Activity 页面变化的相关逻辑。其中核心的 6 种生命周期状态，即 onCreate()、onStart()、onResume()、onPause()、onStop()和 onDestroy()，是 Activity 页面变换的核心，并组成三大生命周期体系，即完整的生命周期、可视的生命周期、前台的生命周期。而剩余的三种生命周期状态，即 onRestart()、onRestoreInstanceState()、onSaveInstanceState()，在特定时机中辅助 Activity 页面变换。这 9 种生命周期状态构成了全部的 Activity 页面生命周期，各种组合保证了页面的正常显示与提供更好的用户体验。

同时，通过一个高级实例，详细地分析三种特殊行为：因硬件导致的生命周期变化、在页面切换时的生命周期变化、系统原因导致的生命周期变化。

1）在因硬件导致的生命周期变化中，主要关注：用户按菜单键，即 Android 手机控制面板的中部按钮导致的生命周期变化；用户按后退键，即 Android 手机控制面板的右侧按钮导致的生命周期变化。

2）在页面切换时生命周期变化中，主要关注：用户跳转非透明页面的常规的生命周期切换；用户跳转透明页面的非常规的生命周期切换，因为底部页面仍然可视，所以不会触发原页面的不可视生命周期。

3）在系统原因导致的生命周期变化中，主要关注：在系统状态改变时，页面保存和恢复状态信息的方式，保证用户体验的一致性，调用 onRestoreInstanceState()的时机。

读者也可以使用本实例，做更多有关于 Android 生命周期的实验，更加完整地掌握 Android 生命周期状态的全部切换样式。

到此，有关 Android 生命周期的全部高级知识，就已完整地呈现在大家面前。希望读者掌握 Android 生命周期的各个细节，也希望继续多多体会，多多练习。

1.2 深入剖析 Activity 的启动模式

在 Android 系统中，Activity 负责展示页面及与用户之间的交互，一个应用的信息流经常需要使用多个页面，如何创建和销毁页面至关重要。有些页面需要多次显示，有些页面需要临时显示。全部 Activity 都会存储在 Activity 栈中，每次创建一个页面，都会向栈中添加，每次销毁一个页面，都会从栈中弹出。频繁地创建与销毁页面，导致系统产生很大的开销。

因此，为了实现 Activity 复用，Android 系统提供 Activity 的启动模式，即 LaunchMode，根据情况选择创建还是复用实例。启动模式主要包括以下四种：Standard（标准模式，默认）、SingleTop（栈顶复用模式）、SingleTask（栈内复用模式）、SingleInstance（单实例模式）。标准模式在每次启动时都会创建 Activity 实例，其他三种单例模式会根据情况选择创建还是复用实例。在 Activity 启动中，创建实例的生命周期为 onCreate() -> onStart() -> onResume()；重用实例的生命周期为 onNewIntent() -> onResume()。在 AndroidManifest 的 Activity 配置中，使用 launchMode 属性，即可设置启动模式，默认是标准模式（standard）；使用 taskAffinity 属性并添加包名，即可设置 Activity 栈，默认是当前包名，此属性仅适用于三种单例（single）模式。

希望通过本节，读者可以更好地理解 Activity 的启动模式（LaunchMode），并且熟练地应用在日常开发中。

本文实例完整代码的下载地址为 https://github.com/SpikeKing/wcl-activity-launchmode-demo。

1.2.1 ADB 命令

ADB 是 Android Debug Bridge 的简称，即 Android 调试桥，是 Android 的调试工具，通过 ADB 的连接，可以在 Shell 中方便地通过 DDMS（Dalvik Debug Monitor Service，Dalvik 调试监控服务）来调试 Android 程序。ADB 的工作方式是监听 Socket TCP 5554 端口，当运行 Android Studio 时，ADB 进程就会自动运行。ADB 是一个客户端-服务器端程序，客户端是开发者的主机，服务器端是连接的 Android 设备。ADB 是 Android SDK 中提供的调试工具，通过该工具支持直接操作 Android 模拟器或者真实设备，其主要功能有：

◎ 在 Shell 中操作设备，如 MonkeyTest、启动或关闭手机等。

◎ 在计算机和模拟器或真实设备之间传输文件，如上传或下载文件。

◎ 在模拟器或真实设备中安装或卸载本地应用的 APK 包。

对于一些常见的 ADB 命令，本节不做过多介绍，主要关注如何查看 Activity 栈的 ADB 命令，即：

```
    adb shell dumpsys activity | sed -n -e '/Stack #/p' -e '/Running activities/,/Run
#0/p'
```

直接获取（dumpsys）的 Activity 信息有些冗余，这里只想关注 Activity 的堆栈信息，因而使用 Shell 的 sed 命令编辑所显示的文字。

◎　-n，表示从选择位置开始显示。

◎　-e，表示多选参数。

◎　-e '/Stack #/p'，输出只含有 Stack #的行。

◎　-e '/Running activities/,/Run #0/p'，输出从 Running activities 至 Run #0 的所有行。

如，输出结果：

```
    Stack #1:
    Running activities (most recent first):
      TaskRecord{9cb774a #121 A=org.wangchenlong.wcl_activity_launchmode_demo U=0
StackId=1 sz=2}
        Run #1: ActivityRecord{79c1a3d u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestAActivity t121}
        Run #0: ActivityRecord{f6ea715 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.MainActivity t121}
    Stack #0:
    Running activities (most recent first):
      TaskRecord{73686bb #117 I=com.google.android.apps.nexuslauncher/.NexusLauncherActivity
U=0 StackId=0 sz=1}
        Run #0: ActivityRecord{4885908 u0 com.google.android.apps.nexuslauncher/.
NexusLauncherActivity t117}
```

通过这个脚本，可以方便地观察 Activity 栈的变化，理解 Standard、SingleTop、SingleTask、SingleInstance 四种启动模式各自的特点。

1.2.2　标准模式

标准模式即 Standard 模式，是启动 Activity 的默认模式，没有特殊配置的 Activity，就会使用标准模式启动。启动 Activity 的基础命令是 startActivity()，参数是 Intent，创建 Intent 需要 Context类与 Activity 的 Class 类。被启动的 Activity 会运行于启动的 Activity 栈，因此启动 Activity 中参数 Intent 的 Context 类，必须使用 Activity 的 Context 启动，如 MainActivity.this，不能使用 Application 的 Context。Application 的 Context 不包含 Activity 栈，Activity 的 Context 才包含 Activity 栈，默认的 Activity 栈是包名，即 TaskRecord 的 A 参数：

```
    TaskRecord{3caa65e3 #2711 A=org.wangchenlong.wcl_activity_launchmode_demo U=0
sz=2}
```

本实例与 Activity 相关，MainActivity 含有两个部分，第一部分是创建 Activity，第二部分是输出 Activity 的生命周期。首页含有两个按钮 mBMyself 和 mBTestA，一个用于创建自己，

即创建重复的当前页面，一个用于创建 TestAActivity，即创建新的页面，调用基本的 Activity 创建方法 startActivity()。使用 ButterKnife 绑定布局 ID，使用 Lambda 表达式简化匿名类的创建。使用 sCount 用于计数，表示 MainActivity 被创建的次数。

```
    public class MainActivity extends AppCompatActivity {
        private static final String TAG = "DEBUG-WCL: " +
MainActivity.class.getSimpleName();
        private static int sCount;
        @Bind(R.id.main_b_myself) Button mBMyself; // 跳转自己
        @Bind(R.id.main_b_test_a) Button mBTestA; // 跳转测试 A
        @Bind(R.id.main_tv_count) TextView mTvCount;

        @Override
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main);
            ButterKnife.bind(this);

            sCount++;
            mTvCount.setText(String.valueOf("当前示例: " + sCount));
            // 不断创建自己的实例
            mBMyself.setOnClickListener(v -> {
                startActivity(new Intent(MainActivity.this, MainActivity.class));
            });
            // 测试类 A
            mBTestA.setOnClickListener(v -> {
                startActivity(new Intent(MainActivity.this, TestAActivity.class));
            });
            Log.e(TAG, "onCreate");
        }
```

为了区分页面是创建还是复用，MainActivity 输出生命周期函数的 Log 信息，即 onCreate()、onNewIntent()、onStart()、onResume()、onPause()、onStop()、onDestroy()。创建 Activity 实例的生命周期是 onCreate()、onStart()、onResume()；复用 Acivity 实例的生命周期是 onNewIntent()、onResume()。通过观察 Activity 的生命周期变化，理解不同启动模式的效果。

```
    @Override protected void onNewIntent(Intent intent) {
        super.onNewIntent(intent);
        Log.e(TAG, "onNewIntent");
    }
    @Override protected void onStart() {
        super.onStart();
        Log.e(TAG, "onStart");
    }
    @Override protected void onResume() {
        super.onResume();
        Log.e(TAG, "onResume");
```

```
    }
    @Override protected void onPause() {
        super.onPause();
        Log.e(TAG, "onPause");
    }
    @Override protected void onStop() {
        super.onStop();
        Log.e(TAG, "onStop");
    }
    @Override protected void onDestroy() {
        super.onDestroy();
        sCount--;
        Log.e(TAG, "onDestroy");
    }
    }
```

标准模式的实验非常简单，即 MainActivity 启动 TestAActivity 即可，TestAActivity 是普通页面，并未添加任何启动模式。通过执行 ADB 显示 Activity 栈的命令，获取 Activity 栈的信息。Stack #0 表示手机桌面的 Activity 栈，本实例在虚拟机中执行，因此栈中有一个 Activity，即 NexusLauncherActivity；Stack #1 表示当前应用的 Activity 栈，是关注的重点，栈（Stack）中含有一个任务记录（TaskRecord），任务记录中含有两个 Activity 记录（ActivityRecord）。整个 Activity 的存储方式是三层结构，即栈（Stack）、任务记录（TaskRecord）、Activity 记录（ActivityRecord），三层分别是一对多的关系，即栈中可以包含多个任务记录，任务记录中可以包含多个 Activity 记录。任务记录中的 A，表示任务记录的包名；任务记录中的 StackId，表示所属栈的 ID；任务记录中的 sz，表示栈中 Activity 记录的数量。在当前应用的任务记录中，存放两个 Activity 记录，即 MainActivity、TestAActivity。TaskRecord 的数据结构是栈，即后进先出，符合用户对于页面的认识顺序，即关掉当前页面返回上一个页面。在此栈中，TestAActivity 位于 MainActivity 之上。

```
    Stack #1:
    Running activities (most recent first):
      TaskRecord{9cb774a #121 A=org.wangchenlong.wcl_activity_launchmode_demo U=0
StackId=1 sz=2}
        Run #1: ActivityRecord{79c1a3d u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestAActivity t121}
        Run #0: ActivityRecord{f6ea715 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.MainActivity t121}
    Stack #0:
    Running activities (most recent first):
      TaskRecord{73686bb #117 I=com.google.android.apps.nexuslauncher/.NexusLauncherActivity
U=0 StackId=0 sz=1}
        Run #0: ActivityRecord{4885908 u0 com.google.android.apps.nexuslauncher/.
NexusLauncherActivity t117}
```

生命周期的顺序是 MainActivity 创建、TestAActivity 创建、MainActivity 销毁，分别调用相

应的生命周期函数。

```
    MainActivity: onCreate
    MainActivity: onStart
    MainActivity: onResume
    MainActivity: onPause
    TestAActivity: onStart
    TestAActivity: onResume
    MainActivity: onStop
```

当用户点击后退键时，TestAActivity 出栈，同时被关闭，随后 MainActivity 位于栈顶，栈中只有一个 Activity，即 MainActivity。

```
    Stack #1:
    Running activities (most recent first):
      TaskRecord{9cb774a #121 A=org.wangchenlong.wcl_activity_launchmode_demo U=0
StackId=1 sz=1}
        Run #0: ActivityRecord{f6ea715 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.MainActivity t121}
```

本例其余的 TestAActivity、TestBActivity、TestCActivity、TestDActivity 与 MainActivity 类似，逻辑中都含有按钮与生命周期两个部分，效果如图 1-8 所示。

图 1-8　MainActivity

1.2.3　栈顶复用模式

栈顶复用模式，即 SingleTop 模式，只有当 Activity 位于栈顶时，再次启动当前的 Activity，复用栈顶的 Activity 实例，不会重新创建，即单例模式；如果位于栈内，与标准模式相同，仍然会重新创建实例。为了验证栈顶复用模式，本例引入 TestBActivity、TestCActivity。其内部逻辑与 TestAActivity 类似，不同的是声明 TestBActivity 的启动模式是栈顶复用模式（SingleTop），具体位于 AndroidManifest 的 Activity 声明中，将 `android:launchMode` 属性设置为 singleTop，其他启动模式的声明方式也类似。

```
    <activity
        android:name=".TestBActivity"
        android:launchMode="singleTop"/>
```

首先，创建标准模式的对比实验，当标准模式 Activity 位于栈顶时，重复创建自己，就会产生多个相同 Activity 实例，如 MainActivity 就是标准模式的 Activity。

```
    Stack #1:
    Running activities (most recent first):
      TaskRecord{440ef8b #124 A=org.wangchenlong.wcl_activity_launchmode_demo U=0
StackId=1 sz=3}
        Run #2: ActivityRecord{ff35662 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.MainActivity t124}
        Run #1: ActivityRecord{49abd5f u0 org.wangchenlong.wcl_activity_launchmode_
demo/.MainActivity t124}
        Run #0: ActivityRecord{a7f1688 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.MainActivity t124}
```

其次，创建栈顶启动栈顶复用模式的实验，操作顺序是：MainActivity 启动 TestAActivity，TestAActivity 启动 TestBActivity，TestBActivity 是栈顶复用模式（SingleTop）的单例，TestBActivity 重复创建自己。观察 TaskRecord 的栈内情况，TestBActivity 即使启动多次，也只有一份实例，在重复启动时，复用栈顶的实例。

```
    Stack #1:
    Running activities (most recent first):
      TaskRecord{440ef8b #124 A=org.wangchenlong.wcl_activity_launchmode_demo U=0
StackId=1 sz=3}
        Run #2: ActivityRecord{d2a2336 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestBActivity t124}
        Run #1: ActivityRecord{2b854e0 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestAActivity t124}
        Run #0: ActivityRecord{a7f1688 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.MainActivity t124}
```

当 TestAActivity 位于栈顶时，创建 TestBActivity（栈顶复用模式），再由 TestBActivity 重复创建自己，TestBActivity 的生命周期由 onStart()和 onResume()变为 onNewIntent()和 onResume()，正常创建的生命周期中含 onStart()，栈顶复用的生命周期中含有 onNewIntent()。

```
    TestAActivity: onPause
    TestBActivity: onStart
    TestBActivity: onResume
    TestAActivity: onStop
    TestBActivity: onPause
    TestBActivity: onNewIntent
    TestBActivity: onResume
```

在本例中，启动模式不同的类，始终是 TestBActivity，通过创建自己或其他方式，介绍不同启动模式之间的差别，如图 1-9 所示。

图 1-9　TestBActivity

最后，创建非栈顶启动栈顶复用模式的实验，当栈顶复用模式的 Activity 位于非栈顶时，其启动与标准模式一样。操作顺序是：MainActivity 启动 TestAActivity，TestAActivity 启动 TestBActivity，TestBActivity 启动 TestCActivity，TestCActivity 再次启动 TestBActivity。TestBActivity 仍是栈顶复用模式的单例。在最后一次启动 TestBActivity 时，由于 TestCActivity 是栈顶，TestCActivity 启动 TestBActivity 不会复用 TestBActivity 的实例，而是重新创建 TestBActivity，栈中保留两份 TestBActivity。

```
    Stack #1:
    Running activities (most recent first):
      TaskRecord{1792f5f0  #2715  A=org.wangchenlong.wcl_activity_launchmode_demo
U=0 sz=5}
       Run #4: ActivityRecord{1e70110b u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestBActivity t2715}
       Run #3: ActivityRecord{c7f4dce u0 org.wangchenlong.wcl_activity_launchmode_
```

```
demo/.TestCActivity t2715}
        Run #2: ActivityRecord{254536cd u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestBActivity t2715}
        Run #1: ActivityRecord{36b2da15 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestAActivity t2715}
        Run #0: ActivityRecord{3a1c4a6a u0 org.wangchenlong.wcl_activity_launchmode_
demo/.MainActivity t2715}
```

栈顶复用模式（SingleTop）是当实例位于栈顶时重复创建，调用 onNewIntent()复用实例；当实例位于栈中时，重复创建，不会复用实例，而是创建新的实例，与标准模式相同，示意图如图 1-10 所示。

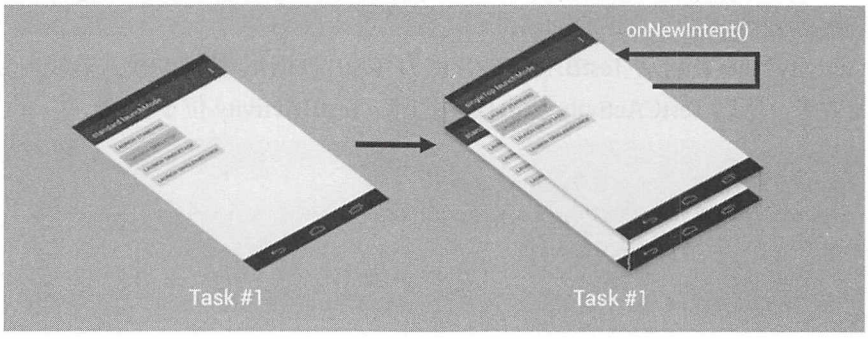


图 1-10　SingleTop

1.2.4　栈内复用模式

栈内复用模式即 SingleTask 模式，只要在栈中存在栈内复用模式的 Activity，无论是在栈顶还是在栈内，多次调用时都会复用实例，不会重新创建，即完全的单例模式，不同于栈顶复用模式。除此之外，栈内复用模式还可以设置不同的任务栈，即 taskAffinity 属性。在 AndroidManifest 中设置 launchMode 属性，TestBActivity 是栈内复用模式（SingleTask）的 Activity。

```
    <activity
        android:name=".TestBActivity"
        android:launchMode="singleTask"/>
```

栈内复用模式包含以下几种情况：

第一种情况，未设置 taskAffinity 属性，任务栈相同。当栈内复用模式的 Activity 实例位于栈顶时，不会重复创建，复用栈顶实例，这一点与栈顶复用模式相同。不同的是，当栈内复用模式的实例位于栈中时，实例重置为栈顶，并清除其上面的其他实例，具有清除顶部（ClearTop）的效果。

操作顺序：MainActivity 启动 TestAActivity，TestAActivity 启动 TestBActivity，TestBActivity

是栈内复用）模式，TestBActivity 启动 TestCActivity，此时，TestBActivity 位于栈中。

```
    Stack #1:
Running activities (most recent first):
  TaskRecord{a0e94f5  #131  A=org.wangchenlong.wcl_activity_launchmode_demo  U=0
StackId=1 sz=4}
    Run  #3:  ActivityRecord{9c049a4  u0  org.wangchenlong.wcl_activity_launchmode_
demo/.TestCActivity t131}
    Run  #2:  ActivityRecord{1eec199  u0  org.wangchenlong.wcl_activity_launchmode_
demo/.TestBActivity t131}
    Run  #1:  ActivityRecord{22bb4ba  u0  org.wangchenlong.wcl_activity_launchmode_
demo/.TestAActivity t131}
    Run  #0:  ActivityRecord{5a28314  u0  org.wangchenlong.wcl_activity_launchmode_
demo/.MainActivity t131}
```

TestCActivity 再重新启动 TestBActivity，因为 TestBActivity 的启动模式是栈内复用，具有清除顶部的效果，所以 TestCActivity 会出栈。此时，TestBActivity 位于栈顶。

```
    Stack #1:
    Running activities (most recent first):
      TaskRecord{a0e94f5 #131 A=org.wangchenlong.wcl_activity_launchmode_demo U=0
StackId=1 sz=3}
        Run #2: ActivityRecord{1eec199 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestBActivity t131}
        Run #1: ActivityRecord{22bb4ba u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestAActivity t131}
        Run #0: ActivityRecord{5a28314 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.MainActivity t131}
```

观察 TestCActivity 与 TestBActivity 的生命周期，TestCActivity 执行关闭的生命周期：onPause() -> onStop() -> onDestroy()；而 TestBActivity 由栈中移至栈顶，依次调用 onNewIntent() -> onStart() -> onResume()。

```
    TestCActivity: onPause
    TestBActivity: onNewIntent
    TestBActivity: onStart
    TestBActivity: onResume
    TestCActivity: onStop
    TestCActivity: onDestroy
```

第二种情况，设置 taskAffinity 属性，任务栈不同，修改 TestBActivity 的 AndroidManifest 配置。taskAffinity 属性适应于栈内复用模式，配合使用，在栈顶复用模式和标准模式中无效。

```
    <activity
        android:name=".TestBActivity"
        android:launchMode="singleTask"
        android:taskAffinity="org.wangchenlong.stack"/>
```

当 taskAffinity 所指定的任务栈不存在时，初次启动栈内复用模式的实例，会创建新的任务

栈，并将实例放置于其中。操作顺序是：MainActivity 启动 TestAActivity，TestAActivity 启动 TestBActivity，TestB 是栈内复用模式且任务栈不同。此时，系统包含两个任务栈，TestBActivity 位于其他任务栈中，注意任务栈的 A 属性不同，一个是 taskAffinity 的属性值，即栈内复用模式的任务栈，一个是应用的包名，即默认的任务栈。

```
    Stack #1:
    Running activities (most recent first):
      TaskRecord{c1349ac #133 A=org.wangchenlong.stack U=0 StackId=1 sz=1}
        Run #2: ActivityRecord{114c451 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestBActivity t133}
      TaskRecord{bb0f675 #132 A=org.wangchenlong.wcl_activity_launchmode_demo U=0
StackId=1 sz=2}
        Run #1: ActivityRecord{18baa1d u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestAActivity t132}
        Run #0: ActivityRecord{90d182c u0 org.wangchenlong.wcl_activity_launchmode_
demo/.MainActivity t132}
```

被启动 Activity 的任务栈，与启动 Activity 的任务栈相同，例如 TestBActivity 启动 TestCActivity，TestBActivity 设置 taskAffinity 属性，属于不同的任务栈，则 TestCActivity 也位于 TestBActivity 所在的任务栈中。操作顺序是：MainActivity 启动 TestAActivity，TestAActivity 启动 TestBActivity（栈内复用模式实例、不同任务栈），TestBActivity 启动 TestCActivity，则 MainActivity 和 TestAActivity 位于相同的任务栈（默认），TestBActivity 和 TestCActivity 位于相同的任务栈（TestBActivity 的 taskAffinity 属性）。此时，栈顶是 TestCActivity。

```
    Stack #1:
    Running activities (most recent first):
      TaskRecord{c1349ac #133 A=org.wangchenlong.stack U=0 StackId=1 sz=2}
        Run #3: ActivityRecord{cd1fcbf u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestCActivity t133}
        Run #2: ActivityRecord{114c451 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestBActivity t133}
      TaskRecord{bb0f675 #132 A=org.wangchenlong.wcl_activity_launchmode_demo U=0
StackId=1 sz=2}
        Run #1: ActivityRecord{18baa1d u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestAActivity t132}
        Run #0: ActivityRecord{90d182c u0 org.wangchenlong.wcl_activity_launchmode_
demo/.MainActivity t132}
```

当任务栈不同时，启动不同栈内的实例，导致任务栈之间的切换，后台任务栈会位于前台。如按手机的 Home 键，然后再次启动应用，则默认任务栈会启动，并不是原有的任务栈，则栈顶元素由 TestCActivity 变为 TestAActivity。

```
    Stack #1:
    Running activities (most recent first):
      TaskRecord{bae21b #137 A=org.wangchenlong.wcl_activity_launchmode_demo U=0
StackId=1 sz=2}
```


```
        Run #3: ActivityRecord{48ccdf0 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestAActivity t137}
      TaskRecord{c58afb8 #138 A=org.wangchenlong.stack U=0 StackId=1 sz=2}
        Run #2: ActivityRecord{fd06649 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestCActivity t138}
        Run #1: ActivityRecord{b89b487 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestBActivity t138}
      TaskRecord{bae21b #137 A=org.wangchenlong.wcl_activity_launchmode_demo U=0
StackId=1 sz=2}
        Run #0: ActivityRecord{a6a708d u0 org.wangchenlong.wcl_activity_launchmode_
demo/.MainActivity t137}
```

当关闭 TestAActivity 时，如按后退键，则下一个页面是与 TestAActivity 相同任务栈的 MainActivity，而不是位于其他任务栈的 TestCActivity。

```
    Stack #1:
    Running activities (most recent first):
      TaskRecord{7c5f433 #141 A=org.wangchenlong.wcl_activity_launchmode_demo U=0
StackId=1 sz=1}
        Run #2: ActivityRecord{bd45264 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.MainActivity t141}
      TaskRecord{222b3f0 #142 A=org.wangchenlong.stack U=0 StackId=1 sz=2}
        Run #1: ActivityRecord{bb074f4 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestCActivity t142}
        Run #0: ActivityRecord{5531a8a u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestBActivity t142}
```

当继续关闭 MainActivity 时，则切换至桌面，并不会切换至 TestCActivity 的任务栈。因为在前面的操作中，单击 Home 键返回桌面，即将桌面任务栈添加至序列中，默认任务栈（包名）的实例全部出栈后，就会切换至桌面任务栈（如 nexuslauncher）。同时，TestCActivity 与 TestBActivity 的任务栈仍会保留在应用中，避免重复创建，符合栈内复用模式。

```
    Stack #0:
    Running activities (most recent first):
      TaskRecord{24206f2      #125      I=com.google.android.apps.nexuslauncher/.
NexusLauncherActivity U=0 StackId=0 sz=1}
        Run #0: ActivityRecord{2a9c957 u0 com.google.android.apps.nexuslauncher/.
NexusLauncherActivity t125}
    Stack #1:
    Running activities (most recent first):
      TaskRecord{222b3f0 #142 A=org.wangchenlong.stack U=0 StackId=1 sz=2}
        Run #1: ActivityRecord{bb074f4 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestCActivity t142}
        Run #0: ActivityRecord{5531a8a u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestBActivity t142}
```

当再次启用应用时，由 MainActivity 启动 TestAActivity，由 TestAActivity 启动 TestBActivity，TestBActivity 的启动模式是栈内复用模式，且仍在内存中，则切换至 TestBActivity 任务栈，并

清除 TestCActivity（清除顶部机制）。

```
    Stack #1:
    Running activities (most recent first):
      TaskRecord{222b3f0 #142 A=org.wangchenlong.stack U=0 StackId=1 sz=1}
        Run #2: ActivityRecord{5531a8a u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestBActivity t142}
      TaskRecord{5a48463 #143 A=org.wangchenlong.wcl_activity_launchmode_demo U=0
StackId=1 sz=2}
        Run #1: ActivityRecord{e5cfa8c u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestAActivity t143}
        Run #0: ActivityRecord{e365d5b u0 org.wangchenlong.wcl_activity_launchmode_
demo/.MainActivity t143}
```

TestAActivity 启动 TestBActivity 的生命周期过程为先关闭 TestCActivity，再复用 TestBActivity，最后停止 TestAActivity。

```
    TestCActivity: onDestroy
    TestAActivity: onPause
    TestBActivity: onNewIntent
    TestBActivity: onStart
    TestBActivity: onResume
    TestAActivity: onStop
```

栈内复用模式是当实例位于栈顶时重复创建，调用 onNewIntent()复用实例，与栈顶复用模式相同；当实例位于栈中时重复创建，同样会复用实例，并且清除至栈顶的全部其他实例，ClearTop 模式。栈内复用模式支持设置 taskAffinity 属性，创建不同的任务栈。栈内复用模式是比较复杂的启动模式，其示意图如图 1-11 所示。

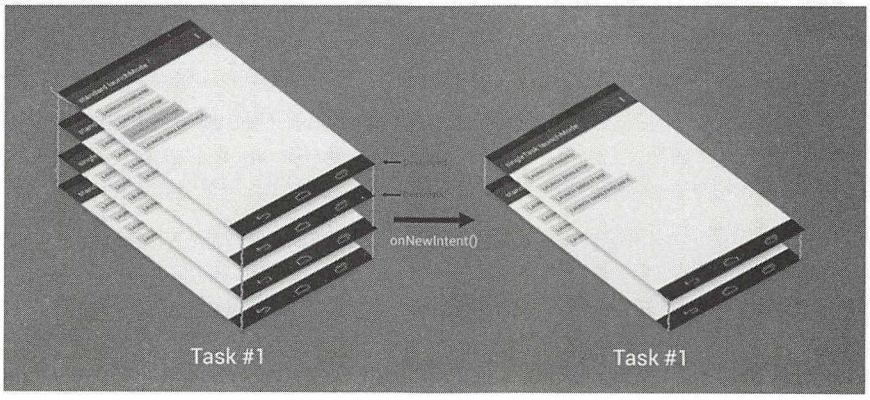


图 1-11　SingleTask

1.2.5 单实例模式

单实例模式（SingleInstance）在启动单实例模式的 Activity 实例时，系统为其创造一个单独的任务栈，以后每次使用时都会使用这个单例，直到其被销毁，属于真正的单例模式。与栈内复用模式不同，栈内复用模式是指定 taskAffinity 属性，所创建的任务栈并不独享，仍可添加其他的 Activity；而单实例模式所创建的任务栈只含有这个实例，所以说是真正的单例模式。将 TestBActivity 的 AndroidManifest 属性的启动模式修改为单实例模式。

```
    <activity
        android:name=".TestBActivity"
        android:launchMode="singleInstance"/>
```

操作顺序：MainActivity 启动 TestAActivity，TestAActivity 启动 TestBActivity，TestBActivity 是单实例模式，TestBActivity 启动 TestCActivity，则 TestCActivity 位于栈顶。在应用内存中，存在两个任务栈，一个是 MainActivity、TestAActivity、TestCActivity，即默认的任务栈，一个是 TestBActivity，即单实例模式的任务栈。两个任务栈的名称相同，都是包名，但是栈的序号不同，为不同任务栈。在本例中，默认任务栈的序号是#144，单实例模式任务栈的序号是#145，也可以使用 taskAffinity 属性，重新为单实例模式的任务栈指定新的包名。因为 TestBActivity 是单实例模式，所创建的任务栈中无法添加 Activity，所以 TestCActivity 仍存储于默认的任务栈中。

```
    Stack #1:
    Running activities (most recent first):
      TaskRecord{b406be #144 A=org.wangchenlong.wcl_activity_launchmode_demo U=0
StackId=1 sz=3}
        Run #3: ActivityRecord{9372e33 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestCActivity t144}
      TaskRecord{ba8f31f #145 A=org.wangchenlong.wcl_activity_launchmode_demo U=0
StackId=1 sz=1}
        Run #2: ActivityRecord{cfa10a8 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestBActivity t145}
      TaskRecord{b406be #144 A=org.wangchenlong.wcl_activity_launchmode_demo U=0
StackId=1 sz=3}
        Run #1: ActivityRecord{bb6ea42 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestAActivity t144}
        Run #0: ActivityRecord{d8053d7 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.MainActivity t144}
```

还有两种后续操作：第一种是关闭当前页面 TestCActivity，第二种是 TestCActivity 再次启动 TestBActivity。首先介绍第一种操作，当关闭 TestCActivity 时，优先出栈当前任务栈的全部实例，再出栈第二层任务栈的全部实例，依次类推。虽然 TestCActivity 由 TestBActivity 启动，但是任务栈不同，如关闭 TestCActivity，则位于栈顶的实例是 TestAActivity，而不是 TestBActivity。接着，关闭 TestAActivity，位于栈顶的实例是 MainActivity，关闭 MainActivity，最终位于栈顶

的实例才是不同任务栈中单实例模式的 TestBActivity。

```
    Stack #1:
    Running activities (most recent first):
      TaskRecord{aae5deb #146 A=org.wangchenlong.wcl_activity_launchmode_demo U=0
StackId=1 sz=2}
        Run #2: ActivityRecord{bfa0f34 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestAActivity t146}
      TaskRecord{b9dc748 #147 A=org.wangchenlong.wcl_activity_launchmode_demo U=0
StackId=1 sz=1}
        Run #1: ActivityRecord{1add81b u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestBActivity t147}
      TaskRecord{aae5deb #146 A=org.wangchenlong.wcl_activity_launchmode_demo U=0
StackId=1 sz=2}
        Run #0: ActivityRecord{9224cb1 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.MainActivity t146}
```

其次介绍第二种操作，当 TestCActivity 再次启动 TestBActivity 时，单实例模式的 TestBActivity 位于栈顶，接着才是默认任务栈中的 MainActivity、TestAActivity、TestCActivity。

```
    Stack #1:
    Running activities (most recent first):
      TaskRecord{3389c55 #149 A=org.wangchenlong.wcl_activity_launchmode_demo U=0
StackId=1 sz=1}
        Run #3: ActivityRecord{a05179b u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestBActivity t149}
      TaskRecord{d7aa66a #148 A=org.wangchenlong.wcl_activity_launchmode_demo U=0
StackId=1 sz=3}
        Run #2: ActivityRecord{7fc07b2 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestCActivity t148}
        Run #1: ActivityRecord{12cd0b4 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestAActivity t148}
        Run #0: ActivityRecord{1129831 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.MainActivity t148}
```

其生命周期函数与其他单例模式类似，都是含有 onNewIntent()。

```
    TestCActivity: onPause
    TestBActivity: onNewIntent
    TestBActivity: onStart
    TestBActivity: onResume
    TestCActivity: onStop
```

在单实例模式下，当创建单实例模式的实例时，单独为其创建一个任务栈，当实例位于栈顶时重复创建，调用 onNewIntent()复用实例，与其他单例模式（栈顶复用模式、栈内复用模式）相同；当实例位于栈中时，重复创建同样会复用实例，将单实例模式的任务栈提前，其示意图如图 1-12 所示。

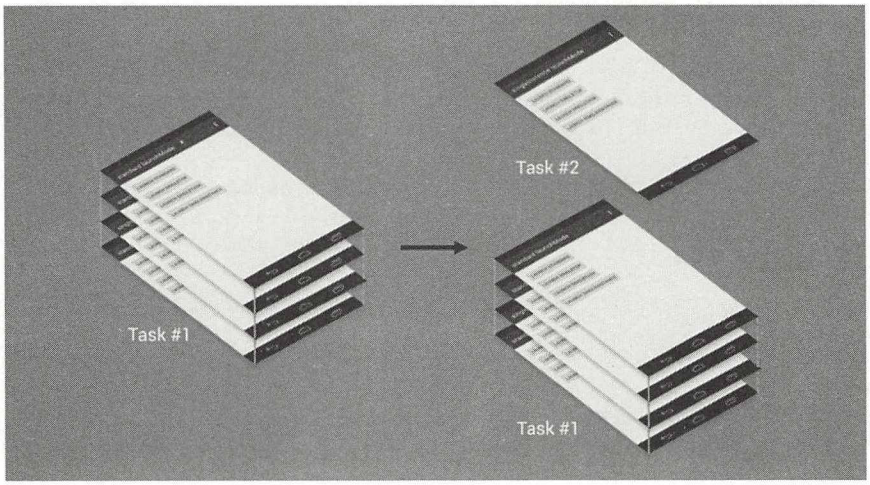


图 1-12　单实例模式

1.2.6　startActivity

除了在 AndroidManifest 中设置 launchMode，修改 Activity 的启动模式，也可以在 startActivity()的 Intent 中设置启动标志位，即 Flag，以修改启动模式。

```
intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
```

常用的标志位如下：

◎ FLAG_ACTIVITY_SINGLE_TOP：同栈顶复用的启动模式。

◎ FLAG_ACTIVITY_NEW_TASK：同栈内复用的启动模式，单独使用不具备清除顶部的效果。

◎ FLAG_ACTIVITY_CLEAR_TOP：如果是栈内复用的启动模式，会清除栈上其他实例，复用实例，调用 onNewIntent()方法；如果是标准的启动模式，即默认模式，则会清除自己和其他实例，再重新创建，调用 onCreate()。

在设置启动模式时，AndroidManifest 与 startActivity()的 Intent 还有一些区别：

◎ AndroidManifest 无法设置清除顶部的效果，只有栈内复用的启动模式含有清除顶部的效果；而 startActivity()的 Intent 支持设置 FLAG_ACTIVITY_CLEAR_TOP 属性，实现清除顶部的效果，并且在标准的启动模式中也可以应用。

◎ startActivity()的 Intent 无法设置单实例的启动模式。

两者任选其一使用即可，如果两种方式均设置，startActivity()的 Intent 的优先级大于 AndroidManifest 的优先级，startActivity()的 Intent 会覆盖 AndroidManifest 的效果。

同时，startActivityForResult()也是启动 Activity 的一种方式，不同于 startActivity()，需要设置返回值。startActivityForResult()使用启动（LaunchMode）模式启动 Activity 时，也会有一些不同，其效果是可以正常传递返回值，但是无法复用实例，即无法触发单例效果，会重复创建多份实例。

操作顺序为 MainActivity 启动 TestAActivity，TestAActivity 启动 TestBActivity，TestBActivity 是栈内复用模式。TestBActivity 使用 startActivityForResult()重复创建自己时，会生成一份新的示例，而 TestBActivity 使用 startActivity()重复创建自己时，会复用默认实例，即产生单例效果。以下是，startActivityForResult()重复创建的任务栈。

```
    Stack #1:
    Running activities (most recent first):
      TaskRecord{8dae57c #116 A=org.wangchenlong.wcl_activity_launchmode_demo U=0
StackId=1 sz=4}
        Run #3: ActivityRecord{48856b3 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestBActivity t116}
        Run #2: ActivityRecord{2e0a535 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestBActivity t116}
        Run #1: ActivityRecord{96bcf27 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.TestAActivity t116}
        Run #0: ActivityRecord{cc6b020 u0 org.wangchenlong.wcl_activity_launchmode_
demo/.MainActivity t116}
```

因为 startActivityForResult()需要返回值，任何时候都需要保留实例，接收返回值，这一特性覆盖原有的单例效果。需要注意的是：在 Android 4.x 以下版本中，通过 startActivityForResult()启动 singleTask，无法正常获取返回值；在 Android 5.x 以上版本中已修复此问题，但是考虑兼容性的问题，不推荐 startActivityForResult()和单例（栈顶复用、栈内复用、单实例）的启动模式同时使用。

在 Android 开发中，Activity 是核心组件，开发者需要认真掌握 Activity 的方方面面，Activity 的启动模式关系到 Activity 在任务栈中的存在形式，影响 Activity 的展示顺序。Android 系统并未提供操作任务栈的接口，无法任意改变 Activity 的存储顺序，栈中 Activity 排列都依赖于启动模式。当用户点击后退时，Activity 出栈；当用户创建页面时，Activity 入栈。Android 系统的任务栈体系分为三层：

- ◎ 第一层是 Stack，每一个应用属于一个 Stack，一一对应。
- ◎ 第二层是 TaskRecord，它含有一组 Activity，通过单例模式（SingleTask 和 SingleInstance）支持创建多个 TaskRecord。
- ◎ 第三层是 ActivityRecord，每一个 Activity 实例就是一个 ActivityRecord，相同的 Activity 支持存在多份实例。

启动模式的本质就是解决如何避免相同的 Activity 创建多个实例。对于标准模式，在每次

启动 Activity 时，都会创建一份实例，在某些情况下，这属性系统资源的浪费，即当重复创建相同的 Activity 时，仅需一份实例即可。因此，Android 系统通过启动模式提供三种解决方案：

- ◎ 栈顶复用模式：当 Activity 位于栈顶时，重复创建实例，直接复用；当 Activity 位于栈中时，与标准模式相同。
- ◎ 栈内复用模式：当 Activity 位于栈顶时，同栈顶复用模式；当 Activity 位于栈中时，清除其上的全部实例，直至位于栈顶。
- ◎ 单实例模式：直接创建一个任务栈，无论 Activity 在任何位置，都会复用这个实例。

需要注意的是，栈内复用模式与单实例模式都支持创建新的任务栈，这会涉及前台任务栈与后台任务栈的关系，即 Stack 中含有多个 TaskRecord。如果使用系统的默认后退操作，即出栈操作，只有当前任务栈全部出栈完毕时，才会切换至其他的任务栈，或者通过启动单例模式的 Activity，才会触发切换任务栈。这个地方很容易操作困惑，即入栈的顺序与出栈的顺序不同。

设置 Activity 的启动模式包含两种方式：一种是在 AndroidManifest 中的 Activity 声明 launchMode 属性，一种是在 startActivity()的 Intent 中添加 Flag 属性；两者各有局限，也可以同时使用，当出现冲突时，startActivity()的 Intent 的设置覆盖 AndroidManifest。最后，注意 startActivityForResult()需要处理返回值，这会导致单例模式的失效，无法复用实例。

在 Android 开发中，当需要重复使用某个页面时，可以选择不同的启动模式，避免重复创建页面，消耗系统资源。只有熟练掌握 Activity 启动模式的各种效果，才能在开发中游刃有余，编写出优雅的代码，成为高级 Android 开发工程师。

1.3 深入剖析 View 的工作流程

在 Android 系统中，视图是最重要的显示控件之一，也是其他显示控件的基类。如图 1-13 所示，在视图的子类中，一类是视图控件组 ViewGroup，用于组合和排列其他控件，另一类是单个视图控件，如 TextView 和 ImageView 等，负责显示特定的视觉样式。无论控件的最终外观如何，都是继承于视图。因此，视图在开发中起着至关重要的作用，是应用与用户之间交流的窗口。

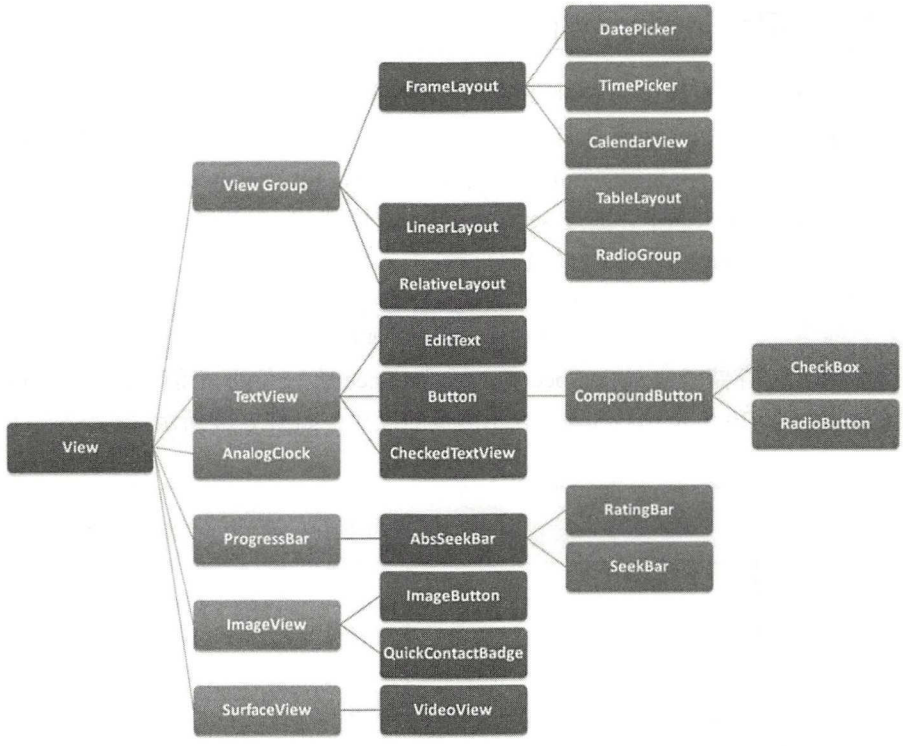


图 1-13　View 的子类集

视图的核心是在页面中展示特定的图像和响应用户的事件。

◎ 视图展示的系统架构，由视图根节点（ViewRoot）开始，视图根节点负责连接窗口管理器（WindowManager）与装饰视图（DecorView），窗口管理器用于响应用户事件，而装饰视图用于展示特定图像。

◎ 视图展示的程序入口，由 performTraversals()方法开始，在方法中依次调用 performMeasure()、performLayout()、performDraw()，即执行测量流程、执行布局流程、执行绘制流程，完成页面的整个绘制流程，将装饰视图中的内容呈现于手机屏幕之上。

如果想要深入剖析视图，需要熟悉视图的工作流程。为了完整地显示不同样式的视图，所有视图的子类都需要执行三大流程，即测量流程、布局流程、绘制流程。在视图由创建到显示的过程中，第一步，测量视图的高度与宽度；第二步，布局视图在父容器中的位置；第三步，绘制视图在屏幕上的显示。

本文实例完整代码参考：Android 源码的 View 和 ViewGroup 类。

1.3.1 装饰视图和 MeasureSpec

在页面 Activity 的布局中，装饰视图就是顶层视图，其核心是 FrameLayout 布局，在 FrameLayout 布局中包含一个竖直方向的 LinearLayout。LinearLayout 布局含有两个部分，一个是标题部分，即 titlebar，一个是内容部分，存放于 android.R.id.content 属性。标题是页面的导航栏，如 ActionBar。在页面中，通过 setContentView()将 android.R.id.content 设置为具体布局，通过如下方法，在 Activity 中获取被设置的布局：

```
ViewGroup content = (ViewGroup) findViewById(android.R.id.content); // 父布局
View view = content.getChildAt(0); // 内容布局
```

视图的具体尺寸存储于 MeasureSpec 中，MeasureSpec 是 32 位的 int 值，高 2 位是 SpecMode，表示测量的模式，低 30 位是 SpecSize，表示测量的尺寸，如图 1-14 所示。

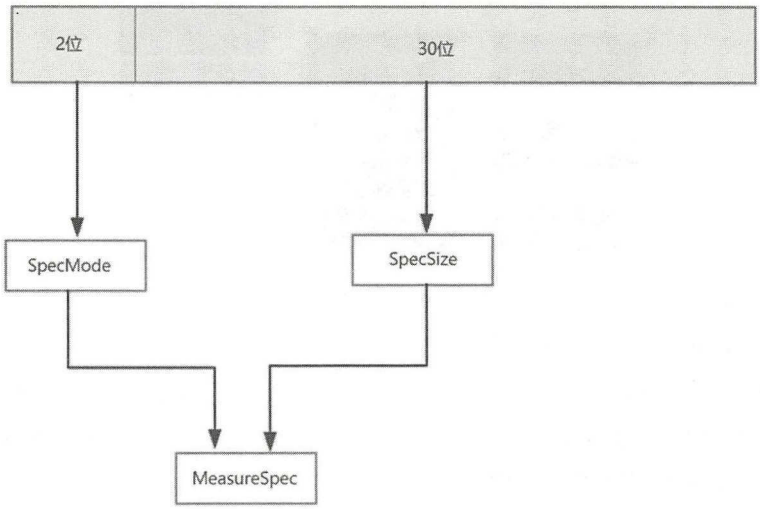


图 1-14 MeasureSpec

在设置过程中，首先选择测量的模式，其次设置测量的尺寸。测量模式含有三种类型，即 UNSPECIFIED、EXACTLY、AT_MOST，其中 UNSPECIFIED 表示不做任何限制；EXACTLY 表示精确尺寸，即 match_parent 和具体数值；AT_MOST 表示不能超过所允许的最大尺寸，即 wrap_content。MeasureSpec 属性被用于测量逻辑中，即 onMeasure()，用于测量 View 的尺寸。在 onMeasure()方法中，

视图的测量尺寸除了根据 MeasureSpec 的属性值，还需要根据父布局的 MeasureSpec 属性值和自身布局参数决定。其中，较为特殊的装饰视图属于根视图，不含有父视图，则由系统的窗口尺寸与自身布局决定。由于视图的层级关系，父视图负责子视图的绘制，因此子视图的具

体尺寸根据父视图的模式和尺寸与自身的模式和尺寸共同决定。

1.3.2　测量

在视图的展示逻辑中，测量是布局和绘制的基础。测量主要负责管理视图的真实高度和宽度，视图不是单独而是组合存在于页面之中。页面的测量由视图组开始，遍历全部子视图或子视图组，再递归汇总，统一管理。具体子视图的测量逻辑在 onMeasure()方法中实现。在自定义视图中，如果子类视图需要修改父类视图的测量值，则需要覆写父类视图的 onMeasure()方法。

示例参考 Android SDK 中的 android.view.View 类。

在 onMeasure()方法中，调用 setMeasuredDimension()设置视图的测量尺寸，参数是测量宽度与测量高度。测量宽度与测量高度调用 getDefaultSize()方法获取默认值，参数是测量的尺寸与模式。测量尺寸在 getSuggestedMinimumWidth()和 getSuggestedMinimumHeight()中获取，测量模式是 widthMeasureSpec 和 heightMeasureSpec。

```
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    setMeasuredDimension(
        getDefaultSize(getSuggestedMinimumWidth(), widthMeasureSpec),
        getDefaultSize(getSuggestedMinimumHeight(), heightMeasureSpec)
    );
}
```

在获取默认尺寸 getDefaultSize()方法中，通过 MeasureSpec 的 getMode()和 getSize()将 MeasureSpec 的模式和尺寸数据分离，根据模式的不同，返回的尺寸数据不同。

◎ UNSPECIFIED 模式未指定模式，直接返回参数 size，即推荐的最小值。宽度最小值为 getSuggestedMinimumWidth()，高度最小值为 getSuggestedMinimumHeight()。

◎ AT_MOST 模式与 EXACTLY 模式相同，都是返回 MeasureSpec 中的尺寸。

其中，AT_MOST 是 wrap_content，尺寸是自身的最小宽度；而 EXACTLY 要么是 match_parent，即父控件支持的最大宽度，要么是一个精确的尺寸数值。

```
public static int getDefaultSize(int size, int measureSpec) {
    int result = size;
    int specMode = MeasureSpec.getMode(measureSpec);
    int specSize = MeasureSpec.getSize(measureSpec);

    switch (specMode) {
        case MeasureSpec.UNSPECIFIED:
            result = size;
            break;
        case MeasureSpec.AT_MOST:
        case MeasureSpec.EXACTLY:
```

```
                result = specSize;
                break;
            }
            return result;
        }
```

在获取建议最小宽度 getSuggestedMinimumWidth()方法中：

◎ 含有背景属性：在已设置最小宽度 mMinWidth 与背景最小宽度 mBackground.getMinimumWidth()两者之间，即 android:minWidth 属性值和 android:background 属性值之间返回最大值。

◎ 不含有背景属性：直接返回已设置最小宽度 mMinWidth，即 android:minWidth 属性值。

```
    protected int getSuggestedMinimumWidth() {
        return (mBackground == null) ? mMinWidth : max(mMinWidth, mBackground.
getMinimumWidth());
    }
```

为了支持视图的 AT_MOST 模式，即 layout 布局中的 wrap_content 属性，需要设置控件的最小尺寸值，否则会默认使用父控件的最大值。导致 AT_MOST 模式无效，而与 EXACTLY 模式相同，也就是说在 layout 布局中，wrap_content 属性与 match_parent 属性的功能相同。

导致这一问题的原因，参考 ViewGroup 源码中的 getChildMeasureSpec()方法，此方法用于设置子视图的属性值。首先，获取测量属性的模式 specMode 和尺寸 specSize，接着获取最大空闲值 size，最后根据不同的模式和不同的布局属性，设置不同的子视图属性值。在 getChildMeasureSpec()中，无论父视图的模式是 EXACTLY 还是 AT_MOST，当子视图的布局属性为 WRAP_CONTENT，测量值均为父布局的最大空闲值 size，即默认值与 match_parent 相同。

```
    public static int getChildMeasureSpec(int spec, int padding, int childDimension)
{
        int specMode = MeasureSpec.getMode(spec);
        int specSize = MeasureSpec.getSize(spec);

        int size = Math.max(0, specSize - padding); // 最大空闲值

        int resultSize = 0;
        int resultMode = 0;

        switch (specMode) {
        // Parent has imposed an exact size on us
        case MeasureSpec.EXACTLY:
            // ...
            } else if (childDimension == LayoutParams.WRAP_CONTENT) {
                // Child wants to determine its own size. It can't be
                // bigger than us.
                resultSize = size; // 父 View 空闲宽度
```

```
            resultMode = MeasureSpec.AT_MOST;
        }
        break;

    // Parent has imposed a maximum size on us
    case MeasureSpec.AT_MOST:
        // ...
        } else if (childDimension == LayoutParams.WRAP_CONTENT) {
            // Child wants to determine its own size. It can't be
            // bigger than us.
            resultSize = size; // 父View空闲宽度
            resultMode = MeasureSpec.AT_MOST;
        }
        break;
    // ...
    }
    return MeasureSpec.makeMeasureSpec(resultSize, resultMode);
}
```

如果需要添加支持 wrap_content 属性，则需要覆写 onMeasure()接口，指定默认的最小宽度 mMinWidth 和最小高度 mMinHeight。在视图的 onMeasure()接口中，在宽度属性 widthMeasureSpec 和高度属性 heightMeasureSpec 中，获取宽度模式和宽度尺寸、高度模式和高度尺寸。当模式为 AT_MOST 时，设置宽度或高度为最小宽度或最小高度，抛弃父类提供的尺寸，即抛弃最大可用尺寸。关于设置 wrap_content 状态下的测量值，也可以参考标准视图 TextView 与 ImageView。

```
private int mMinWidth = 256; // 指定默认最小宽度
private int mMinHeight = 256; // 指定默认最小高度

@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    super.onMeasure(widthMeasureSpec, heightMeasureSpec);
    int widthSpecMode = MeasureSpec.getMode(widthMeasureSpec);
    int widthSpecSize = MeasureSpec.getSize(widthMeasureSpec);
    int heightSpecMode = MeasureSpec.getMode(heightMeasureSpec);
    int heightSpecSize = MeasureSpec.getSize(heightMeasureSpec);
    if (widthSpecMode == MeasureSpec.AT_MOST
            && heightSpecMode == MeasureSpec.AT_MOST) {
        setMeasuredDimension(mMinWidth, mMinHeight);
    } else if (widthSpecMode == MeasureSpec.AT_MOST) {
        setMeasuredDimension(mMinWidth, heightSpecSize);
    } else if (heightSpecMode == MeasureSpec.AT_MOST) {
        setMeasuredDimension(widthSpecSize, mMinHeight);
    }
}
```

在视图组中，不仅需要测量自身，还要需要测量子视图。视图组使用 onMeasure()方法测量

自身，使用 measureChildren()方法递归测量全部子视图。其中，视图组的 onMeasure()方法与视图的类似，measureChildren()方法是视图组独有。在 measureChildren()方法中，首先获取全部子视图数量 mChildrenCount 和全部子视图列表，接着调用 measureChild()方法，遍历全部视图，当视图的可视标签不为移除时，递归绘制子视图，否则放弃绘制子视图。

```
protected void measureChildren(int widthMeasureSpec, int heightMeasureSpec) {
    final int size = mChildrenCount;
    final View[] children = mChildren;
    for (int i = 0; i < size; ++i) {
        final View child = children[i];
        if ((child.mViewFlags & VISIBILITY_MASK) != GONE) {
            measureChild(child, widthMeasureSpec, heightMeasureSpec);
        }
    }
}
```

在测量子视图 measureChild()中，获取视图的布局参数，根据父视图的测量值、左右填充、布局宽度，来计算子视图的测量值，最后调用 measure()方法，完成子视图的测量。

```
protected void measureChild(View child, int parentWidthMeasureSpec,
        int parentHeightMeasureSpec) {
    final LayoutParams lp = child.getLayoutParams();

    final int childWidthMeasureSpec = getChildMeasureSpec(parentWidthMeasureSpec,
            mPaddingLeft + mPaddingRight, lp.width);
    final int childHeightMeasureSpec = getChildMeasureSpec(parentHeightMeasureSpec,
            mPaddingTop + mPaddingBottom, lp.height);

    child.measure(childWidthMeasureSpec, childHeightMeasureSpec);
}
```

除了 measureChildren()方法之外，当测量尺寸涉及边缘时，则需要调用 measureChildWithMargins()方法。与 measureChildren()不同的是，在计算子视图的测量值时，需要额外添加左右边缘值（lp.leftMargin 和 lp.rightMargin）或上下边缘值（lp.topMargin 和 lp.bottomMargin）。最终，仍调用 measure()方法完成子视图的测量。

```
protected void measureChildWithMargins(View child,
        int parentWidthMeasureSpec, int widthUsed,
        int parentHeightMeasureSpec, int heightUsed) {
    final MarginLayoutParams lp = (MarginLayoutParams) child.getLayoutParams();

    final int childWidthMeasureSpec = getChildMeasureSpec(parentWidthMeasureSpec,
            mPaddingLeft + mPaddingRight + lp.leftMargin + lp.rightMargin
                    + widthUsed, lp.width);
    final int childHeightMeasureSpec = getChildMeasureSpec(parentHeightMeasureSpec,
            mPaddingTop + mPaddingBottom + lp.topMargin + lp.bottomMargin
                    + heightUsed, lp.height);
```

```
        child.measure(childWidthMeasureSpec, childHeightMeasureSpec);
    }
```

不同的 ViewGroup 需要实现不同的 onMeasure()方法，如在 LinearLayout、RelativeLayout、FrameLayout 中，实现 onMeasure()的逻辑均不相同。

关于测量值的获取问题也需要注意。视图的测量过程与 Activity 的生命周期并不是同步的，无法在生命周期的方法中获取测量值。测量值只能在异步的测量过程完成后才能获取。两种方法可以获取当前页面的测量值，一种是在 onWindowFocusChanged()方法中，一种是在 onResume()的消息队列队尾。

第一种方法，对于 Activity 的窗口焦点改变 onWindowFocusChanged()方法，在 Activity 窗口获得或失去焦点时都会被调用，也就是在执行 onResume()或 onPause()的过程中会被调用。此时，视图的测量过程（onMeasure()）已经完成，可以获取测量值。首先，获取页面的内容视图 ContentView，在窗口中获取装饰视图，装饰视图的第一个子视图就是布局的根视图 FrameLayout，根视图的第一个子视图就是内容视图 ContentView；接着，在 onWindowFocusChanged()方法中，当页面获得焦点时，即 hasFocus 为 true，在内容视图中，调用 getMeasuredWidth()获取测量宽度，调用 getMeasuredHeight()获取测量高度。

```
    private int mWidth; // 宽度
    private int mHeight; // 高度

    // 在 Activity 获得焦点时，获取测量宽度与高度
    @Override public void onWindowFocusChanged(boolean hasFocus) {
        super.onWindowFocusChanged(hasFocus);
        if (hasFocus) {  // 只在获取焦点时，获取宽度和高度
            mWidth = getContentView().getMeasuredWidth();
            mHeight = getContentView().getMeasuredHeight();
        }
    }

    // 获取 Activity 的 ContentView
    private View getContentView() {
        ViewGroup view = (ViewGroup) getWindow().getDecorView();
        FrameLayout content = (FrameLayout) view.getChildAt(0);
        return content.getChildAt(0);
    }
```

第二种方法与第一种方法类似，也需要获取当前的内容视图 ContentView。在 onResume()中，调用 View 的 post()方法，即在视图的消息队列尾部添加 Runnable 的匿名类。视图的测量过程是在消息队列中完成的，优先完成系统消息，当系统消息全部完成时才会执行用户消息。这样，在异步消息队列的末尾，就能获取页面的测量宽度和测量高度。

```
    private int mWidth; // 宽度
```

```
    private int mHeight; // 高度

    @Override protected void onResume() {
        super.onResume();
        final View view = getContentView();
        view.post(new Runnable() {
            @Override public void run() {
                mWidth = view.getMeasuredWidth();
                mHeight = view.getMeasuredHeight();
            }
        });
    }
```

1.3.3 布局

视图通过布局（layout）确定视图的位置，调用 onLayout()方法。无论是视图还是视图组，其中的布局接口 onLayout()都没有具体逻辑，需要子类实现。视图的 onLayout()方法是空实现，子类可以选择不实现；视图组的 onLayout()方法是抽象方法，子类必须实现。

视图的 onLayout()方法：

```
    protected void onLayout(boolean changed, int left, int top, int right, int bottom)
{}
```

视图组的 onLayout 方法：

```
    @Override
    protected abstract void onLayout(boolean changed, int l, int t, int r, int b);
```

视图的布局开始于 layout()方法。首先，调用 setOpticalFrame()或 setFrame()，确定视图的四个顶点，即 left、right、top、bottom，接着再调用 onLayout()确定子视图的位置。不同的视图和视图组，实现不同的 onLayout()方法。一般而言，在 onLayout()中，都会调用 setChildFrame()，确定子视图的四个顶点，最后子视图再调用 layout()，递归完成整个视图的全部布局过程。

```
    public void layout(int l, int t, int r, int b) {
        // ...
        boolean changed = isLayoutModeOptical(mParent) ?
                setOpticalFrame(l, t, r, b) : setFrame(l, t, r, b);

        if (changed || (mPrivateFlags & PFLAG_LAYOUT_REQUIRED) == PFLAG_LAYOUT_
REQUIRED) {
            onLayout(changed, l, t, r, b);
            // ...
        }
        // ...
    }
```

需要注意的是，在一般情况下，视图的测量的宽高与布局宽高是相同的，但不是绝对的，因为测量和布局的确定时机不同，测量要早于布局完成。如果需要在布局中重新设置宽高，则需要覆写 layout()方法，这样会导致测量的宽高与布局的不同。

```
@Override public void layout(int l, int t, int r, int b) {
    super.layout(l, t, r + 256, b + 256); // 右侧底部，各加 256，无意义
}
```

这样的做法，没有任何实际意义，还会带来误导。如果要修改视图的宽高尺寸，一定要在测量过程中完成。

1.3.4　绘制

在视图的工作流程中，完成测量、布局之后，绘制是最后一步。绘制的流程包含六个步骤，参考视图类 View 的 draw()方法：

◎　第一步，绘制背景，即绘制布局文件中视图的 android:backgroud 属性；

```
if (!dirtyOpaque) {
    drawBackground(canvas);
}
```

◎　第二步，存储画布的层次，为颜色渐变做准备，设置渐变颜色接口 setFadeColor()；

```
int solidColor = getSolidColor();
if (solidColor == 0) {
    final int flags = Canvas.HAS_ALPHA_LAYER_SAVE_FLAG;
    if (drawTop) {
        canvas.saveLayer(left, top, right, top + length, null, flags);
    }
    if (drawBottom) {
        canvas.saveLayer(left, bottom - length, right, bottom, null, flags);
    }
    if (drawLeft) {
        canvas.saveLayer(left, top, left + length, bottom, null, flags);
    }
    if (drawRight) {
        canvas.saveLayer(right - length, top, right, bottom, null, flags);
    }
} else {
    scrollabilityCache.setFadeColor(solidColor);
}
```

◎　第三步，绘制视图的内容，调用 onDraw()，onDraw()是抽象接口，需要视图子类实现；

```
if (!dirtyOpaque) onDraw(canvas);
```

◎　第四步，绘制全部子视图的内容，调用 dispatchDraw()，同样的，dispatchDraw()

也是抽象接口，需要视图子类实现；

```
dispatchDraw(canvas);
```

◎ 第五步，绘制渐变边缘，同时恢复层次，即绘制完内容，再绘制边缘；

```
if (drawRight) {
    matrix.setScale(1, fadeHeight * rightFadeStrength);
    matrix.postRotate(90);
    matrix.postTranslate(right, top);
    fade.setLocalMatrix(matrix);
    p.setShader(fade);
    canvas.drawRect(right - length, top, right, bottom, p);
}

canvas.restoreToCount(saveCount);
```

◎ 第六步，最后一步，绘制装饰内容，如标记、滚动条等。

```
onDrawForeground(canvas);

public void onDrawForeground(Canvas canvas) {
    onDrawScrollIndicators(canvas);
    onDrawScrollBars(canvas);
    // ...
}
```

因此，绘制流程大致分为：绘制背景 -> 绘制自身内容 -> 绘制子视图 -> 绘制前置内容，即 drawBackground() -> onDraw() -> dispatchDraw() -> onDrawForeground()。

视图的工作流程拆分为三大流程，即测量流程、布局流程、绘制流程。因此，视图的核心逻辑就是测量、布局、绘制。测量调用 onMeasure()接口、布局调用 onLayout()接口、绘制调用 onDraw()接口。其中，有两点需要强调一下：

◎ 视图的尺寸：在测量中，调用 getMeasuredHeight()与 getMeasuredWidth()，获取已测量完成的高度与宽度；在布局中，调用 getHeight()与 getWidth()获取已布局完成的高度与宽度。需要注意的是，因时机不同，测量的度量可能不同于布局的度量。

◎ 子视图的绘制：对于子视图而言，测量和布局分别调用 onMeasure()与 onLayout()绘制视图自身，同时遍历调用子视图的相应方法；而绘制将视图自身与子视图分离，使用 onDraw()绘制视图自身，使用 dispatchDraw()绘制子视图。

视图的三大工作流程将视图的展示逻辑解耦，测量关注于视图自身的尺寸，布局关注于视图之间的组合，绘制关注于视图展示的细节。三个工作流程提供丰富的对外接口，子类可以继承接口扩展功能。一般而言，以 on 为前缀的方法都是支持继承覆写的，运行于非 on 方法中。非 on 方法保留逻辑的不变性，on 方法支持逻辑的可变性，实现高内聚低耦合的程序设计思想，也属于模板方法的设计模式。通过源码学习开发技巧和理解程序逻辑很有必要，也是高级程序

员必备的基本素养之一。

1.4　深入剖析 View 的动画原理

在应用开发中，动画效果可以为应用增添不少颜色，也可以提升用户体验。其中，视图作为 Android 系统的核心展示原件，也具有一些自由的动画效果，主要分为两类，即视图动画和属性动画，如图 1-15 所示。视图动画以图片变换为主，其效果包括平移（Translate）动画、缩放（Scale）动画、旋转（Rotate）动画、透明（Alpha）动画等。属性动画以改变属性为主，通过动态地改变视图的某些属性值，达到动画效果。

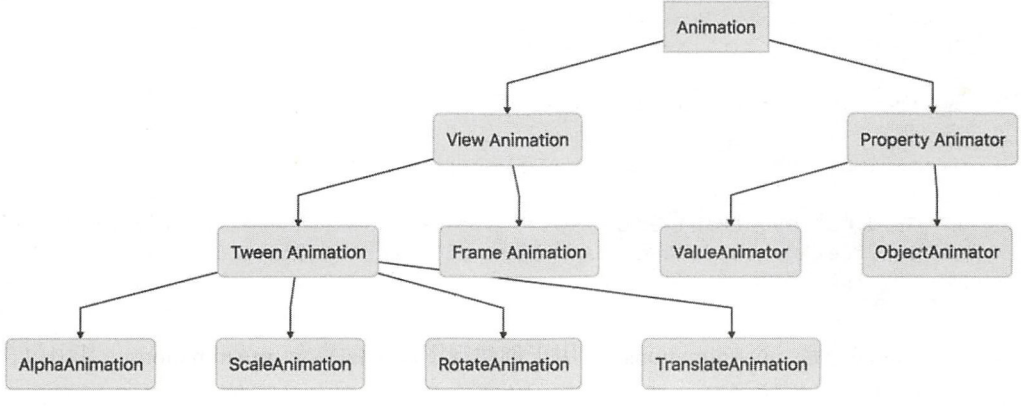


图 1-15　动画

◎　视图动画：分为补间（Tween）动画和帧（Frame）动画，其中补间动画主要应用于页面启动渐入渐出的动画、页面进入和退出的动画、对话框底部弹出的动画等；而帧动画主要应用于网络请求的进度等待、下拉刷新的进度等待等，需要多幅图片的组合使用。

◎　属性动画：分为值（Value）动画和对象（Object）动画，两者都是通过动态地改变属性值，实现动画效果，主要应用于圆形进度条的按百分比移动效果、控件背景颜色的闪烁切换效果等。

如果想要真正地理解动画原理，还需要从代码入手。本例将基于图片视图动态地变换填充图片，逐个展示不同动画的实现方法。

本文实例完整代码的下载地址为 https://github.com/SpikeKing/AnimationPlayer。

1.4.1 默认视图动画

本例从一个简单的 HelloWorld 工程开始，将布局 activity_main.xml 内容替换为列表视图控件，即 RecyclerView。

```
    <android.support.v7.widget.RecyclerView
        android:id="@+id/main_rv_grid"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
```

在页面 MainActivity 的入口 onCreate()函数中，首先，设置布局文件 setContentView()和绑定布局属性 ButterKnife.bind()，其次，初始化动画效果组合 initAnimations()，最后，设置列表布局样式 mRvGrid.setLayoutManager()和设置列表适配器 mRvGrid.setAdapter()。

```
    @Override protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ButterKnife.bind(this);  // 绑定布局 ID

        mContext = getApplicationContext();  // 上下文

        initAnimations(mContext);  // 创建动画组合

        mRvGrid.setLayoutManager(new GridLayoutManager(mContext, 2)); // 每行两列
        mRvGrid.setAdapter(new GridAdapter(mContext, mAnimations, mFrame, mTexts,
mImages));  // 设置适配器
    }
```

关于动画的部分是 initAnimations()，用于创建视图动画的集合 mAnimations，其中包含平移动画、缩放动画、旋转动画、透明动画、动画合集、自定义动画等。默认动画都是调用 AnimationUtils 的 loadAnimation()方法，创建动画 Animation 类，参数是页面信息 Context 和动画文件 anim，而自定义动画是通过继承动画 Animation 类的 Rotate3dAnimation 类，重写类中的接口，实现不同的动画效果。

```
    private void initAnimations(Context context) {
        mAnimations = new ArrayList<>();
        mAnimations.add(AnimationUtils.loadAnimation(context,
R.anim.anim_translate));  // 平移动画
        mAnimations.add(AnimationUtils.loadAnimation(context, R.anim.anim_scale));
// 缩放动画
        mAnimations.add(AnimationUtils.loadAnimation(context,
R.anim.anim_rotate));  // 旋转动画
        mAnimations.add(AnimationUtils.loadAnimation(context, R.anim.anim_alpha));
// 透明动画
        mAnimations.add(AnimationUtils.loadAnimation(context,  R.anim.anim_all));
// 动画合集
```

```
        final Rotate3dAnimation anim = new Rotate3dAnimation(0.0f, 720.0f, 100.0f,
100.0f, 0.0f, false);
        anim.setDuration(2000);
        mAnimations.add(anim);  // 自定义动画
    }
```

平移动画：即 R.anim.anim_translate，其中的属性 duration 表示动画的持续时间；fromXDelta 表示 X 轴的起始坐标；fromYDelta 表示 Y 轴的起始坐标；toXDelta 表示 X 轴的终止坐标；toYDelta 表示 Y 轴的终止坐标；fillAfter 表示当动画完成后的停留状态，如在平移之后，位置不复原；interpolator 表示动画变换的插值器，如加速移动。最终的动画效果是从视图的右上角开始，缓慢地滑动至中间位置。

```
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:fillAfter="true"
    android:interpolator="@android:anim/accelerate_interpolator">
    <!--平移动画-->
    <translate
        android:duration="2000"
        android:fromXDelta="50"
        android:fromYDelta="-100"
        android:toXDelta="0"
        android:toYDelta="0"/>
</set>
```

缩放动画：即 R.anim.anim_scale，其中的属性，duration 表示动画的持续时间；fromXScale 表示宽度的起始比例；fromYScale 表示高度的起始比例；toXScale 表示宽度的终止比例；toYScale 表示高度的终止比例。最终的动画效果是从视图的左上角位置开始，缓慢地放大至整个区域。

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <!--缩放动画-->
    <scale
        android:duration="2000"
        android:fromXScale="0.0"
        android:fromYScale="0.0"
        android:toXScale="1.0"
        android:toYScale="1.0"/>
</set>
```

旋转动画：即 R.anim.anim_rotate，其中的属性 duration 表示动画的持续时间；fromDegrees 表示旋转角度的起始值；toDegrees 表示旋转角度的终止值；pivotX 表示旋转中心 X 轴的坐标；pivotY 表示旋转中心 Y 轴的坐标。同样地，fillAfter 表示是否保留最终状态。最终的动画效果是从视图的中心位置开始，逆时针旋转 720°（2 圈）。

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:fillAfter="true">
```

```
    <!--旋转动画-->
    <rotate
        android:duration="2000"
        android:fromDegrees="0"
        android:pivotX="50%"
        android:pivotY="50%"
        android:toDegrees="-720"/>
</set>
```

透明动画：即 R.anim.anim_alpha，其中的属性 duration 表示动画的持续时间；fromAlpha 表示透明度的起始值，toAlpha 表示透明度的终止值。最终的动画效果是图像从透明状态变换为非透明状态。

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <!--透明动画-->
    <alpha
        android:duration="2000"
        android:fromAlpha="0.1"
        android:toAlpha="1.0"/>
</set>
```

组合动画：即 R.anim.anim_all，即融合平移、缩放、旋转、透明等四种动画的效果。最终的动画效果是图像旋转着从视图的左上角位置，旋转着滚入屏幕，并且逐渐变大变清晰。

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
     android:duration="2000">
    <!--平移动画-->
    <translate
        android:fromXDelta="50"
        android:fromYDelta="-100"
        android:toXDelta="0"
        android:toYDelta="0"/>

    <!--缩放动画-->
    <scale
        android:duration="2000"
        android:fromXScale="0.0"
        android:fromYScale="0.0"
        android:toXScale="1.0"
        android:toYScale="1.0"/>

    <!--旋转动画-->
    <rotate
        android:duration="2000"
        android:fromDegrees="0"
        android:pivotX="50%"
        android:pivotY="50%"
```

```
        android:toDegrees="-720"/>

    <!--透明动画-->
    <alpha
        android:duration="2000"
        android:fromAlpha="0.1"
        android:toAlpha="1.0"/>
</set>
```

至此，四种常用的默认动画效果，即平移动画、缩放动画、旋转动画、透明动画，已经全部介绍。通过基础动画的组合，也可以创建更多类型的动画效果。

1.4.2　自定义视图动画

本例的自定义视图动画类是 Rotate3dAnimation，即旋转三维动画。Rotate3dAnimation 类继承于动画基类 Animation，重写初始化动画方法 initialize()和应用转换方法 applyTransformation()。

Rotate3dAnimation 的构造器含有六个参数，fromDegrees 是旋转的起始角度，toDegrees 是旋转的终止角度，centerX 是旋转中心的 X 轴，centerY 是旋转中心的 Y 轴，depthZ 是转动的 Z 轴，reverse 是逆时针方向或顺时针方向。

```
public Rotate3dAnimation(
        float fromDegrees, float toDegrees,
        float centerX, float centerY,
        float depthZ, boolean reverse) {
    mFromDegrees = fromDegrees;  // 起始角度
    mToDegrees = toDegrees;  // 目标角度
    mCenterX = centerX;  // 旋转中心的 X 轴
    mCenterY = centerY;  // 旋转中心的 Y 轴
    mDepthZ = depthZ;  // 深度 Z 轴
    mReverse = reverse; // 是否反转
}
```

在初始化 initialize()方法中，创建照相类 Camera，用于实现复制的动画效果。

```
@Override public void initialize(int width, int height, int parentWidth, int
parentHeight) {
    super.initialize(width, height, parentWidth, parentHeight);
    mCamera = new Camera();
}
```

动画类 Animation 的核心方法是 applyTransformation()，用于实现定制的动画效果。

◎　计算旋转的起始角度 fromDegrees 和旋转的终止角度 degrees。通过差值时间 interpolatedTime，获取角度的转动差值，计算最终的角度值。

◎　获取转动中心的位置，X 轴是 centerX，Y 轴是 centerY。

◎ 获取照相类 Camera，获取转换矩阵 Matrix。

◎ 保存当前的照相状态，即 camera.save()，用于修改属性。

◎ 根据旋转方向 mReverse，设置转动 Z 轴的方向，逆时针或者顺时针。

◎ 设置照相类的 Y 轴旋转角度，即 camera.rotateY(degrees)；设置照相类的视图矩阵，与转换矩阵相同，即 camera.getMatrix(matrix)；重新恢复照相类的状态，即 camera.restore()。

◎ 设置视图矩阵的转换中心点，转换前的中心调用 preTranslate()，转换后的中心调用 postTranslate()。

◎ 继续执行父类的相关操作，即调用 super.applyTransformation()。

```
    @Override protected void applyTransformation(float interpolatedTime,
Transformation t) {
        final float fromDegrees = mFromDegrees;
        float degrees = fromDegrees + ((mToDegrees - fromDegrees) * interpolatedTime);
// 结尾度数

        // 中心点
        final float centerX = mCenterX;
        final float centerY = mCenterY;

        final Camera camera = mCamera;
        final Matrix matrix = t.getMatrix();

        camera.save(); // 照相机

        // Z 轴平移
        if (mReverse) {
            camera.translate(0.0f, 0.0f, mDepthZ * interpolatedTime);
        } else {
            camera.translate(0.0f, 0.0f, mDepthZ * (1.0f - interpolatedTime));
        }

        camera.rotateY(degrees); // Y 轴旋转
        camera.getMatrix(matrix);
        camera.restore();

        // 围绕 View 的中心点进行旋转
        matrix.preTranslate(-centerX, -centerY);
        matrix.postTranslate(centerX, centerX);

        super.applyTransformation(interpolatedTime, t);
    }
```

在主页 MainActivity 类的 initAnimations()中，实例化自定义动画类，设置构造器相关参数，并且设置动画持续时间。最终的动画效果是图像沿着左侧的 Y 轴，顺时针旋转 720°（2 圈），

最终保持原位。

```
    final Rotate3dAnimation anim = new Rotate3dAnimation(0.0f, 720.0f, 100.0f, 100.0f,
0.0f, false);
    anim.setDuration(2000);
```

视图动画的展示接口是调用视图的 setAnimation()方法，将设置完成的动画类 Animation 放入视图的动画属性中，当首次填充视图时，自动执行被设置的动画属性。若再次执行动画，由于视图已经填充完毕，则需要调用视图的 startAnimation()方法，才能再次执行。

```
    holder.getImageView().setAnimation(mAnimations.get(position));
    holder.getButton().setOnClickListener(new View.OnClickListener() {
        @Override public void onClick(View v) {
            holder.getImageView().startAnimation(mAnimations.get(position));
        }
    });
```

这六种视图动画的样式，即平移动画、缩放动画、旋转动画、透明动画、组合动画、自定义动画的效果如图 1-16 所示。

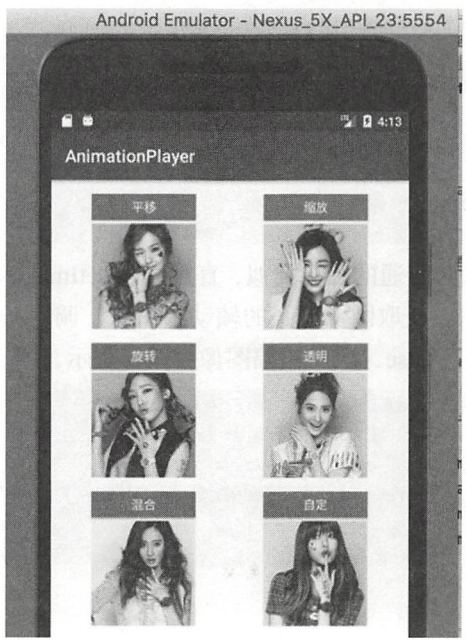


图 1-16　视图动画

1.4.3 帧动画

除了视图动画，还有一种非常简单的动画形式，就是帧动画。帧动画的含义是一帧一帧地播放动画，是动画的最基本形式，通过特定的帧图像实现动画效果。

```
private @DrawableRes int mFrames = R.drawable.anim_images; // 帧动画图像
```

本例的帧动画是 R.drawable.anim_images。根属性是 animation-list，即动画帧列表，其中的 oneshot 属性，当值为 true 时，只执行一次，当值为 false 时，循环执行。在列表中包含三个图像，drawable 表示图像资源，duration 表示图像持续时间。最终的动画效果是每隔 250 毫秒（1/4 秒），切换一次图像，三幅图像共切换三次，不断循环。

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:oneshot="false">
    <!--循环显示三个图片-->
    <item
        android:drawable="@drawable/seo_square"
        android:duration="250"/>
    <item
        android:drawable="@drawable/kim_square"
        android:duration="250"/>
    <item
        android:drawable="@drawable/sunny_square"
        android:duration="250"/>
</animation-list>
```

帧动画的图像设置方法与普通的图像类似，直接通过 setImageResource()方法设置图像资源即可。当启动帧动画时，需要获取图像视图的帧动画图像，调用 start()方法即可启动。因为动画图像的 oneshot 属性设置为 false，所以三幅图像会循环展示，不会终止。

```
holder.getImageView().setImageResource(mFrame);
holder.getButton().setOnClickListener(new View.OnClickListener() {
    @Override public void onClick(View v) {
        ((AnimationDrawable) holder.getImageView().getDrawable()).start();
    }
});
```

1.4.4 属性动画

视图动画的样式较为固定，只提供几类基础的动画形式，所有组合动画都是基于这些基础动画，可扩展性并不强，因此 Android 系统又添加属性动画作为补充。属性动画仅支持 Android API 11 及以上版本，如果版本较低，则需要使用 nineoldandroids 的第三方动画支持库，网址为

http://nineoldandroids.com/。

属性动画的核心是通过动态地改变对象属性，实现动画效果，因而属性必须提供可访问接口，即含有 set()和 get()方法，支持调用。如果实现动态地改变视图属性的效果，可以使用包装器模式或插值模式。包装器模式对应的是对象动画（ObjectAnimator），而插值模式对应的是值动画（ValueAnimator）。

包装器模式（Wrapper 模式）基于对象动画，通过 Wrapper 包装类，将视图封装，为视图的特定属性提供 set()与 get()接口，用于对象动画的调用。本例封装视图的宽度属性，即 width，提供 setWidth()用于设置宽度和 getWidth()用于获取宽度。宽度值来源于视图的布局参数 LayoutParams，当重新设置参数时，需要调用视图的 requestLayout()方法，重绘布局。

```
private static class ViewWrapper {
    private View mView;

    public ViewWrapper(View view) {
        mView = view;
    }

    @SuppressWarnings("unused")
    public int getWidth() {
        return mView.getLayoutParams().width;
    }

    @SuppressWarnings("unused")
    public void setWidth(int width) {
        mView.getLayoutParams().width = width;
        mView.requestLayout();
    }
}
```

关于容易产生混淆的两个视图重绘方法，requestLayout()和 invalidate()，区别如下：

◎ requestLayout()：当视图确定自身不再适合现有区域时，调用 requestLayout()方法，要求父视图重新调用 onMeasure()和 onLayout()，重新设置当前视图的位置。特别地，当视图的布局参数 LayoutParams 发生改变时，而且改变的值还未应用于视图之上，这时比较适合调用 requestLayout()方法重新加载布局参数。

◎ invalidate()：调用 invalidate()方法，导致视图本身重绘，不会影响当前视图与父视图或其他视图的位置关系。

将视图放入视图包装器中，对象动画通过视图包装器的 get()和 set()方法动态地修改宽度属性值，值的范围由参数 start 和 end 传入，持续时间为 2000 毫秒（2 秒），最后调用 start()方法，播放动画效果。

```
private void performWrapperAnimation(final View view, final int start, final int
```

```
end) {
        ViewWrapper vw = new ViewWrapper(view);
        ObjectAnimator.ofInt(vw, "width", start, end).setDuration(2000).start(); //
启动动画
    }
```

将图像视图、起始属性值、终止属性值传入 performWrapperAnimation()方法，即可启动动画。同时，设置点击按钮事件，也可以触发播放动画的效果。最终的动画效果是图像由中心点缓慢放大，直至填充图像控件。

```
    performWrapperAnimation(holder.getImageView(), 0, Utils.dp2px(mContext, 120));
    holder.getButton().setOnClickListener(new View.OnClickListener() {
        @Override public void onClick(View v) {
            performWrapperAnimation(holder.getImageView(), 0, Utils.dp2px(mContext,
120));
        }
    });
```

注意，布局参数 LayoutParams 的数值是 PX 像素，在使用时，如果以 DP 为单位，则需要由 DP 转换至 PX，其转换方法如下：

```
    public static int dp2px(Context context, int dp) {
        final float scale = context.getResources().getDisplayMetrics().density;
        return (int) (dp * scale + 0.5f);
    }
```

插值模式：基于值动画（ValueAnimator），其实现方法如下：

◎ 创建值动画（ValueAnimator）实例，调用 ofInt()设置值的范围为 1~100，即执行 100 次循环。

◎ 调用 addUpdateListener()设置值动画的更新监听，在监听的匿名类中，创建整型估值器（IntEvaluator），用于根据进度比例（fraction）和范围，计算偏移值。

◎ 覆写动画更新接口 onAnimationUpdate()，调用 getAnimatedFraction()，获取当前值动画 ValueAnimator 的比例值 fraction。

◎ 使用整型估值器 IntEvaluator，根据比例（fraction）、起始值（start）、终止值（end）等三个值，计算当前布局的宽度。

◎ 调用视图的重绘布局方法 requestLayout()，完成布局的重绘。

◎ 值动画 ValueAnimator 类，根据值的范围，如 1~100（100 次），不断地更新内部值、不断地调用 onAnimationUpdate()监听、不断地修改视图的布局宽度、不断地重绘视图，从而达到动画的效果。

最终的动画效果，与包装器模式类似，都是将图片逐渐放大，效果相同。

```
    private void performListenerAnimation(final View view, final int start, final
int end) {
        ValueAnimator valueAnimator = ValueAnimator.ofInt(1, 100);
```

```
        valueAnimator.addUpdateListener(new ValueAnimator.AnimatorUpdateListener()
{
            // 持有一个 IntEvaluator 对象，方便下面估值的时候使用
            private IntEvaluator mEvaluator = new IntEvaluator();

            @Override
            public void onAnimationUpdate(ValueAnimator animator) {
                // 获得当前动画的进度值，整型，1-100 之间
                int currentValue = (Integer) animator.getAnimatedValue();

                // 获得当前进度占整个动画过程的比例，浮点型，0-1 之间
                float fraction = animator.getAnimatedFraction();
                // 直接调用整型估值器通过比例计算出宽度，然后再设给 Button
                view.getLayoutParams().width = mEvaluator.evaluate(fraction, start,
end);
                view.requestLayout();
            }
        });
        valueAnimator.setDuration(2000).start();
    }
```

插值模式动画的调用方式几乎与包装器模式相同，只是将调用接口由performWrapperAnimation()换为 performListenerAnimation()。实现的效果也是默认启动就会播放动画效果，点击顶部按钮也可以播放。

```
    performListenerAnimation(holder.getImageView(),    0,    Utils.dp2px(mContext,
120));
    holder.getButton().setOnClickListener(new View.OnClickListener() {
        @Override public void onClick(View v) {
            performListenerAnimation(holder.getImageView(),                    0,
Utils.dp2px(mContext, 120));
        }
    });
```

本例使用属性动画的两种方式，即包装器模式和插值模式，实现相同的动画效果，开发者可以自由选择熟悉的方法。其中，包装器模式更侧重于由对象动画类 ObjectAnimator，控制动画的变化效果；而插值模式则侧重于不断地回调值动画类 ValueAnimator 的动画更新接口 onAnimationUpdate()，在接口中，不断地重绘视图布局，达到动画的变化效果。因而，包装器模式更简单，更依赖于系统提供的动画对象；插值模式稍微复杂，需要编写较多的动画逻辑，可扩充性会更强。

在列表控件中，帧动画、包装器模式动画、插值模式动画，都类似于视图动画的展示方式，由按钮和图像控制组件，其效果如图 1-17 所示。

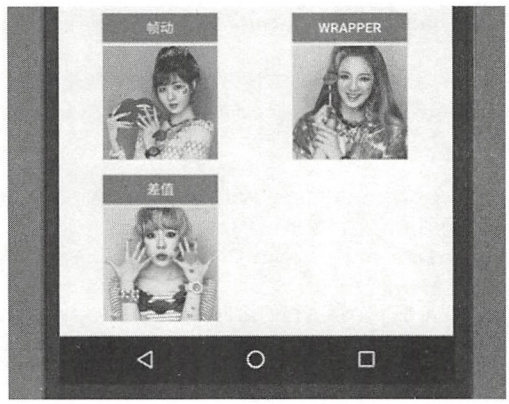


图 1-17　帧动画和属性动画

1.4.5　列表控件

本例通过一个列表控件 RecyclerList，将全部动画效果组合起来，并动态地展示。列表控件的每项由顶部的动画播放按钮和底部的动画展示控件组成。

列表布局 ID（main_rv_grid），通过 ButterKnife 的@BindView 注解，绑定至 RecyclerView 的实例 mRvGrid。

```
    @BindView(R.id.main_rv_grid) RecyclerView mRvGrid;
```

列表控件的布局样式 LayoutManager 是每行两个的网格样式 GridLayoutManager；列表控件的适配器 Adapter 是自定义的网格适配器 GridAdapter。

```
    mRvGrid.setLayoutManager(new GridLayoutManager(mContext, 2)); // 每行两列
    mRvGrid.setAdapter(new GridAdapter(mContext, mAnimations, mFrames, mTexts,
mImages)); // 设置适配器
```

网格适配器 GridViewHolder 使用标准的 RecyclerView 适配器开发范式，继承于 RecyclerView.Adapter，重写构造器方法 GridAdapter()、ViewHolder 创建方法 onCreateViewHolder()、ViewHolder 绑定方法 onBindViewHolder()。

在 onCreateViewHolder()中，将列表项布局 item_anim 填充为视图，再由视图创建列表项的 ViewHolder。ViewHolder 用于管理填充的数据和复用布局的结构。

```
    @Override public GridViewHolder onCreateViewHolder(ViewGroup parent, int
viewType) {
        View view =
LayoutInflater.from(parent.getContext()).inflate(R.layout.item_anim, parent,
false);
        return new GridViewHolder(view);
    }
```

在 onBindViewHolder()中，首先，通过 setAnimation()设置列表项的入场动画，接着，根据不同的列表项位置创建不同的动画效果：

◎ 列表项第 1~6 项，展示六种不同的视图动画，即平移、缩放、旋转、透明、混合、自定义等动画。

◎ 列表项第 7 项，展示帧动画。

◎ 列表项第 8 项，展示包装器模式的属性动画。

◎ 列表项第 9 项，展示插值模式的属性动画。

通过不同动画形式的对比，理解不同动画的实现方式。

```
    @Override public void onBindViewHolder(final GridViewHolder holder, final int
position) {
        setAnimation(holder.getContainer(), position);  // 设置左侧滑动的动画效果

        holder.getImageView().setImageResource(mImages[position]);
        holder.getButton().setText(mTexts[position]);

        switch (position) {
            case 6: // 使用帧动画
                holder.getImageView().setImageResource(mFrame);
                holder.getButton().setOnClickListener(new View.OnClickListener() {
                    @Override public void onClick(View v) {
                        ((AnimationDrawable)
holder.getImageView().getDrawable()).start();
                    }
                });
                break;
            case 7: // 使用 Wrapper 属性动画
                performWrapperAnimation(holder.getImageView(), 0, Utils.dp2px(mContext,
120));
                holder.getButton().setOnClickListener(new View.OnClickListener() {
                    @Override public void onClick(View v) {
                        performWrapperAnimation(holder.getImageView(),          0,
Utils.dp2px(mContext, 120));
                    }
                });
                break;
            case 8: // 使用插值属性动画
                performListenerAnimation(holder.getImageView(), 0, Utils.dp2px(mContext,
120));
                holder.getButton().setOnClickListener(new View.OnClickListener() {
                    @Override public void onClick(View v) {
                        performListenerAnimation(holder.getImageView(), 0, Utils.dp2px
(mContext, 120));
                    }
                });
```

```
                break;
            default:
                holder.getImageView().setAnimation(mAnimations.get(position));
                holder.getButton().setOnClickListener(new View.OnClickListener() {
                    @Override public void onClick(View v) {
                        holder.getImageView().startAnimation(mAnimations.get
(position));
                    }
                });
                break;
        }
    }
```

在 setAnimation()中，设置每一个列表项的入场动画，动画形式是左侧滑入，属于 Android SDK 中自带的动画效果，即 android.R.anim.slide_in_left，持续时间为 3000 毫秒（3 秒）。mLastPosition 用于控制列表项动画，只在创建时执行一次，而在重建时不执行。

```
    private void setAnimation(View viewToAnimate, int position) {
        if (position > mLastPosition || mLastPosition == -1) {
            Animation    animation    =    AnimationUtils.loadAnimation(mContext,
android.R.anim.slide_in_left);
            animation.setDuration(3000);
            viewToAnimate.startAnimation(animation);
            mLastPosition = position;
        }
    }
```

最终的网格布局效果如图 1-18 所示。

在 Android 应用的开发过程中，常常需要利用动画效果提升用户体验，突出核心功能，但是需要注意性能指标，大量的动画效果容易导致内存占用过大。

在动画效果的选择方面，优先选择视图动画，即 ViewAnimation，毕竟视图动画适配较低的系统版本，同时容易开发与实现，然而视图动画的局限性就是动画效果较少，加上各种动画的组合，也无法满足特定的动画效果。这时，就需要使用高级的属性动画，即 PropertyAnimation。

属性动画的两种实现方式各有特色：包装器模式通过对象动画器自动地执行动画；插值模式通过值动画器的回调函数手动地执行动画，是包装器模式的复杂版本。当只针对控件的单一属性实现动画效果时，使用包装器模式比较简单和方便；当需要控件的多个属性联动实现动画效果时，使用插值模式比较易于控制。

同时，需要注意的是，有些动画效果的执行单位是以像素为基准，如果需要在不同分辨率中实现相同效果，需要将常用的测量单位 DP 转换为像素 PX。

动画效果是应用开发的难点，也是最具想象力的部分，熟练掌握各种动画技巧有助于开发出优质的应用，成为合格的高级 Android 工程师。

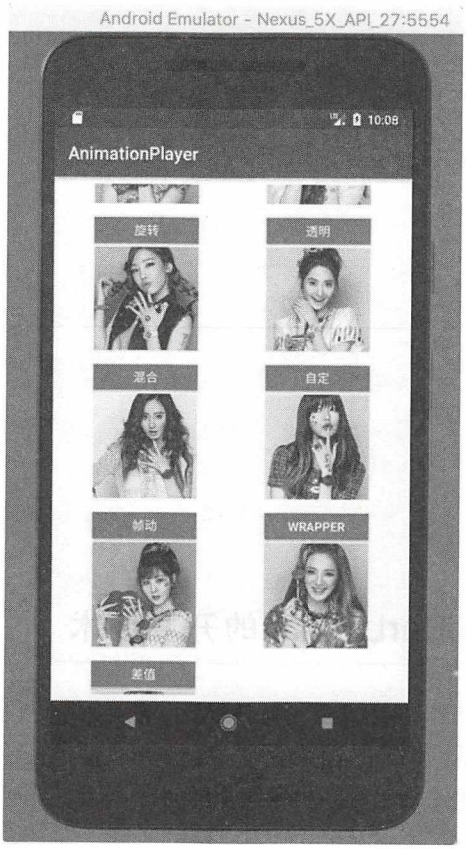

Android Emulator - Nexus_5X_API_27:5554
10:08
AnimationPlayer
旋转
透明
混合
自定
帧动
WRAPPER
差值

图 1-18　网格布局

第 2 章
高阶控件

2.1 熟练掌握 AppBarLayout 的开发技术

Material Design 简称 MD，中文简译为材料设计，是 Android 设计师团队为用户创造出的全新视觉设计语言。它既遵循了经典的设计原则，也汲取了流行的科技元素，具有颠覆性的创新理念。MD 通过构建系统化的动效和利用合理化的空间，融合其他设计理念，创建出与众不同的触感，而这一新的设计理念来源于设计师对纸墨的研究。随着科技的进步，这种理念也会不断地发展和创新。

在设计范式中，触摸实体的表面和边缘会提供类似真实效果的触觉体验，熟悉的触感会让用户快速地理解和使用；触摸实体的多样性也可以呈现出更多反映真实世界的设计效果，同时抽象的效果绝不会脱离客观的物理规律。核心在于光效、表面质感、运动质感三方面，分别是解释物体运动规律、交互方式、空间关系的关键。真实的光效可模拟物体之间的交合关系、空间关系和单个物体的运动。

在 MD 中，所有控件都是基于这种设计理念，其中光效、表面质感、运动质感时刻贯穿着 MD 控件的设计与开发。作为在 MD 中最核心的几种控件之一，AppBarLayout 布局也符合这一设计理念，并且拥有绚丽的动画效果。AppBarLayout 布局通过内容驱动，更加平滑地切换页面；通过滑动手势，减少生硬的点击访问；通过图片展示，便捷地传达页面所要表述的内容。因此，AppBarLayout 布局作为 Android 的高级控件，也是 Android 高级开发者的“必备武器”之一，熟练掌握大有裨益。本节将通过实例帮助大家掌握 AppBarLayout 布局的全部开发技术。

本节实例的完整代码下载地址为 https://github.com/SpikeKing/TestAppBar。

如图 2-1 所示为本实例的最终效果，看到这么精美的布局样式，你一定想知道这是如何实现的吧？

图 2-1　本实例的最终效果

2.1.1　搭建项目框架

为了完整地呈现出 AppBarLayout 的开发流程，我们在 Android Studio（简称 AS）中创建一个新的项目作为开始。既然 AppBarLayout 与 MD 的设计理念有关，本节就选择一个新的工程样式 Navigation Drawer Activity（导航抽屉）作为项目的基础模板，为大家提供更加丰富的选择空间，因为在这个 Activity 页面中也包含若干个符合 MD 理念的其他控件，如图 2-2 所示。

在选择 Navigation Drawer Activity 作为工程的 MainActivity 时，在 MainActivity 中就会含有若干已经编写好的模板代码，我们可以根据自己喜好修改这些项目属性。比如，修改工程的主题颜色，主题颜色的修改也会导致系统控件颜色的修改，若干系统控件会默认使用这些颜色。colorPrimary 表示主题的基础颜色；colorPrimaryDark 表示主题的较深基础颜色；colorAccent 表示主题的强调颜色。因为本实例的部分展示内容是关于少女时代（Girl's Generation，简称 GG，一个韩国的流行演唱团体）的若干信息，所以把主题颜色设置为粉色系，即 colorPrimary 属性

和 colorPrimaryDark 属性均为#FF1493，colorAccent 属性为 FF4081。

```
<resources>
    <color name="colorPrimary">#FF1493</color>
    <color name="colorPrimaryDark">#FF1493</color>
    <color name="colorAccent">#FF4081</color>
</resources>
```

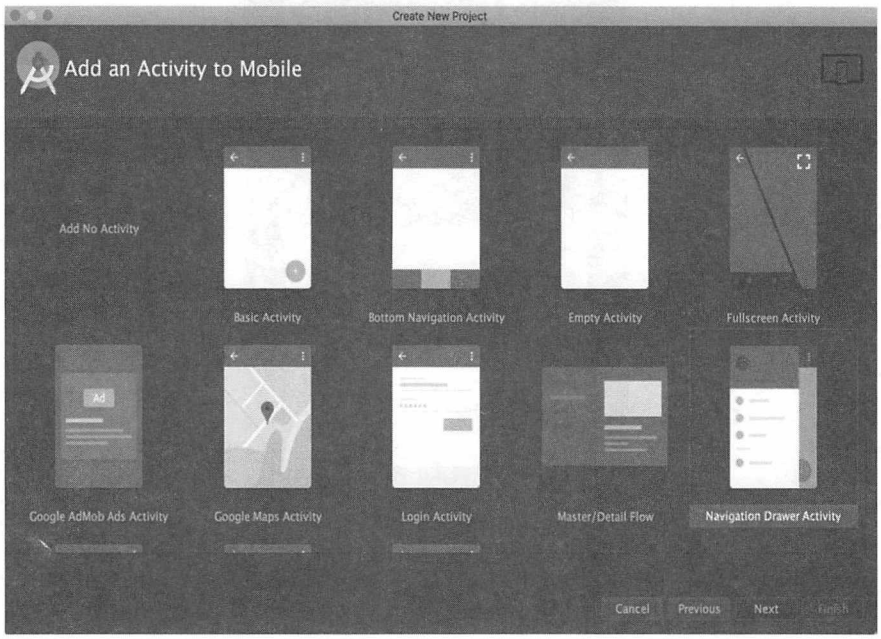


图 2-2　Navigation Drawer Activity

导航抽屉（Navigation Drawer）示例工程的核心就是抽屉布局，同样支持定制，比如修改抽屉布局的主题颜色，将顶部渐变颜色组修改成粉色渐变系。因此，需要修改抽屉布局的颜色渐变文件 side_nav_bar.xml，将起始颜色 startColor、中部颜色 centerColor、尾部颜色 endColor 分别修改为不同的粉色系颜色值，同时颜色变换使用线性变换，角度设置为 135°。

```
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle" >
    <gradient
        android:startColor="#FF34"
        android:centerColor="#FF3E96"
        android:endColor="#FF1493"
        android:type="linear"
        android:angle="135"/>
</shape>
```


最终，修改后的颜色效果如图 2-3 所示。

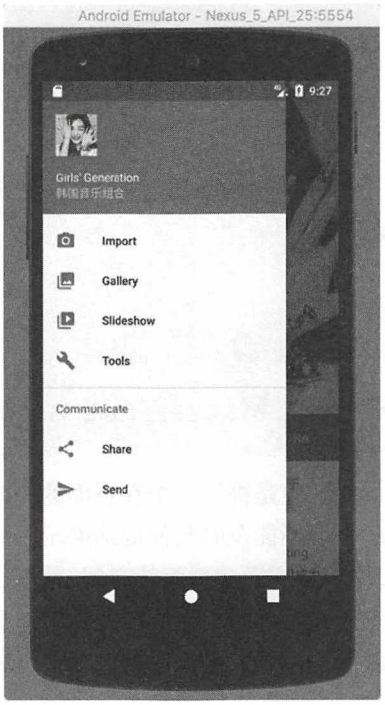


图 2-3　修改后的颜色效果

MD 设计模式既可以带来更多的视觉体验，也可以提供大量的个性化定制，让开发者拥有更多的选择，这是 MD 为开发者带来的炫酷体验之一。因此，我们也可以通过定制其他的主题颜色，改变其他控件的默认颜色系。

打开项目的源码，观察一下 MainActivity.java 的默认生成代码，由于使用导航抽屉的主题样式，所以在源码中会额外增加一些关于导航抽屉部分的代码。如添加导航栏控件 Toolbar 放置于页面的顶部，一般用于添加用户常用的操作，如后退按钮或消息提示中心等，或用于显示页面的标题，类似于 ActionBar；添加抽屉布局 DrawerLayout，在 ActionBarDrawerToggle 中设置抽屉布局的属性，调用触发器 Toggle 的 syncState 方法，同步状态。添加 NavigationView 导航视图，不同于 Toolbar 控件，NavigationView 是与 DrawerLayout 关联的控件，在抽屉顶部显示一些重要的信息，如在本例中，NavigationView 显示 GG 某成员的头像和 GG 的组合简介。

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    ButterKnife.bind(this);
```

```
        // 导航栏
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        // 抽屉布局
        DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
        ActionBarDrawerToggle toggle = new ActionBarDrawerToggle(
                this, drawer, toolbar, R.string.navigation_drawer_open, R.string.
navigation_drawer_close);
        drawer.setDrawerListener(toggle);
        toggle.syncState();

        // 抽屉布局中的导航视图
        NavigationView navigationView = (NavigationView) findViewById(R.id.nav_
view);
        navigationView.setNavigationItemSelectedListener(this);
    }
```

MD 设计者们创造抽屉布局的初衷是提供一个在页面的侧面进行交互的功能，因为以往只能通过 Tab 键左右滑动，重要的页面不能及时与便捷地展示，所以需要跨越过多个页面。而在页面的左侧添加抽屉布局之后，用户只需滑动页面的左侧，就可以快速地进入该页面进行交互，降低了摩擦成本。但是，缺点也是显而易见的，这样做会增加页面模块的复杂度，使页面略显凌乱，导致应用的主题并不突出等。设计本是仁者见仁、智者见智的问题，增加一个新的控件供开发者选择，这终归是好的。

注意：关于 ToolBar，具有开发经验的读者可能会疑惑。在 MD 以前已经存在导航栏，即 ActionBar，为什么 MD 设计者们又提出一个名为 Toolbar 的类似布局呢？因为在早期的版本里，ActionBar 已经绑定到 Activity 中，是独立于页面布局（ContentView）存在的，与 Activity 紧密地耦合在一起，难于定制，也无法与其他控件联动，如本例中的 AppBarLayout 需要将导航栏显示或隐藏，使用 ActionBar 无法实现。这也是多数公司都会使用定制视图代替 ActionBar，模拟导航栏的功能，禁用系统自带的 ActionBar 的原因。为了将导航栏与 Activity 完全解耦，使其重归页面布局的怀抱，MD 设计者们创建了 Toolbar，完全代替 ActionBar 的功能，通过 Toolbar 实现多控件之间的联动与动画效果的增强，为用户带来更佳的操作体验。

2.1.2　页面设置 ViewPager 布局

为了更全面地分析 AppBarLayout 布局的一些知识点，本例将 AppBarLayout 布局与 ViewPager 布局融合在一起，既可以上下滚动（AppBarLayout），也可以左右滚动（ViewPager），动态地展示页面信息，也为读者展示 AppBarLayout 在复杂情况中的应用。

首先，在 MainActivity 的主布局 activity_main.xml 中，添加 layout 布局标签，在 app_bar_main.xml 中编写主要布局逻辑。其中，DrawerLayout 与 NavigationView 都是导航抽屉主题默认生成的布局样式。DrawerLayout 是抽屉布局整个布局样式的根布局，装载一个抽屉内容布局和一个主页内容布局，抽屉内容在 NavigationView 中。NavigationView 是抽屉内容布局，包含两个部分，一个是头部，即 headerLayout；一个是底部，即 menu。NavigationView 的头部用于展示抽屉中的某些信息，使用单独布局@layout，内部可以包含图片控件和文字控件；NavigationView 的底部用于展示可选的菜单项，默认是菜单资源，即@menu，支持用户点击菜单的某一选项，进行跳转。

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.widget.DrawerLayout
    android:id="@+id/drawer_layout"
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:openDrawer="start">

    <!--引入 AppBarLayout 的主布局-->
    <include
        layout="@layout/app_bar_main"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>

    <!--DrawerLayout 默认导航视图-->
    <android.support.design.widget.NavigationView
        android:id="@+id/nav_view"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:fitsSystemWindows="true"
        app:headerLayout="@layout/nav_header_main"
        app:menu="@menu/activity_main_drawer"/>

</android.support.v4.widget.DrawerLayout>
```

接着，编写 AppBarLayout 的核心布局 app_bar_main.xml。在根布局 CoordinatorLayout 中添

加 NestedScrollView 和 ViewPager 两个控件。NestedScrollView 是嵌套上下滚动控件，由于 AppBarLayout 的主要功能是上下滚动、展示和关闭图片，底部的内容也需要配合上下滚动实现联动，因此 NestedScrollView 需要添加在根布局 CoordinatorLayout 中。NestedScrollView 的另一个优势是嵌套功能，即 Nested，可以兼容其他滚动控件，如 ViewPager，这样页面既支持上下滑动，又支持左右滑动，并且防止滑动冲突的产生。ViewPager 放置在 NestedScrollView 中，实现左右切换页面的功能，通过滑动即可切换至下一个页面。

在布局中的 android:fillViewport 属性，负责管理 NestedScrollView 内部组件是否填充，设置为 true 表示完全填充，设置为 false 表示非填充。例如在本例中，当 fillViewport 的值为 true 时，ViewPager 布局会被完全填充到 NestedScrollView 中。同时，在布局中的 app:layout_behavior 属性负责管理布局的联调样式。在 App 命名空间中的属性属于自定义属性，属性的值设置为 @string/appbar_scrolling_view_behavior，在 AppBarLayout 的逻辑中根据不同的属性值，设置不同的联动效果。

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:context=".MainActivity">

    <!--上下滚动的 NestedScrollView，内部包含 ViewPager-->
    <android.support.v4.widget.NestedScrollView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:fillViewport="true"
        android:scrollbars="none"
        app:layout_behavior="@string/appbar_scrolling_view_behavior">

        <android.support.v4.view.ViewPager
            android:id="@+id/main_vp_container"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            app:layout_behavior="@string/appbar_scrolling_view_behavior"/>

    </android.support.v4.widget.NestedScrollView>

</android.support.design.widget.CoordinatorLayout>
```

最后，在 MainActivity 的 onCreate()方法中初始化 ViewPager，将布局关联至逻辑。ViewPager 主要包含两个部分：一个是适配器（Adapter），一个是滑动监听器（PageChangeListener）。

ViewPager 的适配器用于管理切换的页面，根据不同的滑动位置显示对应的页面。ViewPager 的滑动监听器的作用是当 ViewPager 进行滑动时，会在监听器中回调相应的接口，开发者根据业务需求，在接口中编写相应的逻辑，掌控滑动过程的变化。需要注意的是，在配置完基本的 ViewPager 功能后，调用 TabLayout 的 setupWithViewPager()方法，将 ViewPager 关联至 TabLayout，TabLayout 根据 ViewPager 的 Adapter 的特定接口，创建相应的 Tab 组。同时，它们也会产生联动效果，即当滑动 ViewPager 的页面时，Toolbar 的 Tab 标签会随之切换；反之，当切换 TabLayout 的 Tab 标签时，ViewPager 的页面也会随之滑动。在 MD 的设计样式中，控件之间的联动是非常重要的一环。

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // ...

        // 设置 ViewPager 布局
        SimpleAdapter adapter = new SimpleAdapter(getSupportFragmentManager());
        mVpContainer.setAdapter(adapter);

mVpContainer.addOnPageChangeListener(PagerChangeListener.newInstance(adapter,
mIvTarget, mIvOutgoing));
        mTlTab.setupWithViewPager(mVpContainer); // 注意在 TabLayout 中关联 ViewPager
    }
```

Activity 页面设置 ViewPager 的部分已经结束，下面我们接着详细地分析一下 ViewPager 的具体配置。ViewPager 包含两个部分，第一部分是在 ViewPager 中展示的 Fragment 页面，第二部分是控制 ViewPager 显示的适配器（Adapter）。

第一部分主要介绍 Fragment 页面的逻辑。由于 ViewPager 是页面组合空间，其需要展示多个 Fragment 页面，本例的重点在于讲解 AppBarLayout 的开发技术，因此复用 Fragment 的逻辑，通过不同的参数，创建多个不同的 Fragment。TEXTS 是全部显示信息的列表，根据序号（selectionNum）的不同，TextView 显示的内容不同。onCreateView()方法负责创建 Fragment 布局，onViewCreated()方法负责在 Fragment 布局中显示内容，两者都是 Fragment 的核心方法。SimpleFragment 的功能是根据不同的序号显示不同的文本，在本例中显示不同的 GG 组合成员的自我介绍信息。

> 注意：对于需要传递参数（Argument）的 Fragment，这里有一个小技巧，就是通过静态接口封装类的创建。通过静态接口，规范化 Fragment 的创建，类的创建者仅仅需要关注传递的参数，而不必考虑代码的细节。静态接口负责将参数打包，储存在 Argument 对象中，传递给类使用。

```
    public class SimpleFragment extends Fragment {
        private static final String ARG_SELECTION_NUM = "arg_selection_num";
        // 参数的 Tag

        // 显示的文本信息
        private static final int[] TEXTS = {
                R.string.tiffany_text,
                R.string.taeyeon_text,
                R.string.yoona_text
        };

        @Bind(R.id.main_tv_text) TextView mTvText;

        public SimpleFragment() {
        }

        /**
         * 通过静态接口创建 Fragment，规范参数的使用
         *
         * @param selectionNum 参数
         * @return 创建的 Fragment
         */
        public static SimpleFragment newInstance(int selectionNum) {
            SimpleFragment simpleFragment = new SimpleFragment();
            Bundle args = new Bundle();
            args.putInt(ARG_SELECTION_NUM, selectionNum);
            simpleFragment.setArguments(args);
            return simpleFragment;
        }


        @Nullable
        @Override
        public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup
container, @Nullable Bundle savedInstanceState) {
            View view = inflater.inflate(R.layout.fragment_main, container, false);
            ButterKnife.bind(this, view);
            return view;
        }

        @Override
        public void onViewCreated(View view, @Nullable Bundle savedInstanceState) {
            super.onViewCreated(view, savedInstanceState);
            mTvText.setText(TEXTS[getArguments().getInt(ARG_SELECTION_NUM)]);
            // 设置文本信息
        }
    }
```


第二部分主要介绍 ViewPager 适配器的逻辑。本例使用 ViewPager 的基本功能，适配器直接继承于 FragmentPagerAdapter，设置默认的构造器，传递 Activity 的 FragmentManager 参数。FragmentManager 负责将 Fragment 附着于 Activity 中，这个附着逻辑属于适配器的内部机制。适配器含有若干重要方法，控制在 ViewPager 中 Fragment 的各种状态。如 getItem()方法，根据当前 ViewPager 的滑动位置，显示对应的 Fragment；getCount()方法，根据所包含 Fragment 的数量，设置 ViewPager 的页数；getPageTitle()方法，根据每个页面展示的主题不同，设置页面的标题，如果 ViewPager 与 TabLayout 联动，则在 TabLayout 中，每个 Tab 的名称也使用 getPageTitle()方法的返回值。

在适配器中，设置名为 Section 的数据结构，储存字符串(String)信息与图片资源((???) int)信息，用于储存每个 GG 组合成员的名字和照片。并且，在 Session 数据结构中填充多组数据，构成静态数组，作为储存信息的集合，以便在 Fragment 布局中显示这些信息。同时，又设置一个公有接口 getDrawable()，供外部根据位置获取位置对应的图片资源信息，在本例中外部是 AppBarLayout 的顶部图片控件。

注意：在本例中，使用注解的方式声明图片资源属性 int 值，即@DrawableRes，这样就强制要求编译器在编译阶段中检查所传递的 int 值是否为图片资源，以提高代码的安全性。

```
public class SimpleAdapter extends FragmentPagerAdapter {

    // 展示信息
    private static final Section[] SECTIONS = {
            new Section("Tiffany", R.drawable.tiffany),
            new Section("Taeyeon", R.drawable.taeyeon),
            new Section("Yoona", R.drawable.yoona)
    };

    // 默认构造器
    public SimpleAdapter(FragmentManager fm) {
        super(fm);
    }

    // 根据不同的位置 (position)，显示不同的 Fragment
    @Override
    public Fragment getItem(int position) {
        return SimpleFragment.newInstance(position);
    }

    // 子页面 Fragment 的个数
    @Override
    public int getCount() {
        return SECTIONS.length;
```

```
    }

    //当 TabLayout 联动时，每个页面的标题为 Tab 的标题
    @Override
    public CharSequence getPageTitle(int position) {
        if (position >= 0 && position < SECTIONS.length) {
            return SECTIONS[position].getTitle();
        }
        return null;
    }

    // 图片接口
    public @DrawableRes int getDrawable(int position) {
        if (position >= 0 && position < SECTIONS.length) {
            return SECTIONS[position].getDrawable();
        }
        return -1;
    }

    // 存储类
    private static final class Section {
        private final String mTitle; // 标题
        private final @DrawableRes int mDrawable; // 图片

        public Section(String title, int drawable) {
            mTitle = title;
            mDrawable = drawable;
        }

        public String getTitle() {
            return mTitle;
        }

        public @DrawableRes int getDrawable() {
            return mDrawable;
        }
    }
}
```

设置 ViewPager 之后的效果如图 2-4 所示。

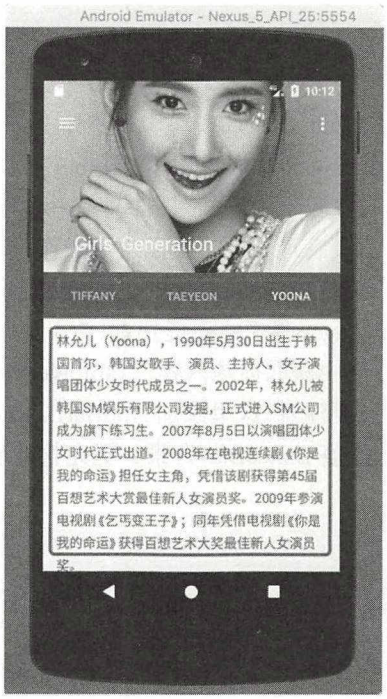


图 2-4　设置 ViewPager 之后的效果

2.1.3　页面添加 AppBarLayout 布局

下面是本章的核心：在页面中添加 AppBarLayout 布局。首先设置资源文件，位置在前文提到的布局文件 app_bar_main.xml 中。在布局的上半部分添加 AppBarLayout，在布局的下半部分是前文提到的 NestedScrollView 和 ViewPager。

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:context=".MainActivity">

    <!--AppBarLayout 布局-->
    <android.support.design.widget.AppBarLayout...>

    <!--NestedScrollView 布局-->
```


```
    <android.support.v4.widget.NestedScrollView...>

</android.support.design.widget.CoordinatorLayout>
```

本节主要关注的是 AppBarLayout 的布局模块。从整体上分析，AppBarLayout 的框架包含两个控件布局：CollapsingToolbarLayout 布局（坍塌 Toolbar 布局）和 TabLayout 布局。顾名思义，CollapsingToolbarLayout 布局来源于 Toolbar 控件，是一种特殊的 Toolbar 控件，Toolbar 控件的基础样式就是顶部导航栏，负责管理若干个页面常用操作；同样地，CollapsingToolbarLayout 也是一种特殊的顶部导航栏，导航栏随着用户的上下滑动操作，同步扩大或缩小整体显示。TabLayout 布局，在上节讲解 ViewPager 时简略地提到过，用于显示 ViewPager 不同页面的标题，从 ViewPager 的 getPageTitle()方法中获取标题内容，TabLayout 调用 setupWithViewPager()方法关联 ViewPager，两者配合使用。还要注意，这里添加 AppBarLayout 的 android:fitsSystemWindows 属性，并设置为 true，使其完全覆盖 StatusBar 背景，更完整地展示在 CollapsingToolbarLayout 布局中的整个图片。

```
<android.support.design.widget.AppBarLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:fitsSystemWindows="true"
    android:theme="@style/AppTheme.AppBarOverlay">

    <!--CollapsingToolbarLayout，坍塌 Toolbar 布局-->
    <android.support.design.widget.CollapsingToolbarLayout...>

    <!--TabLayout 布局，用于与 ViewPager 的联动-->
    <android.support.design.widget.TabLayout...>

</android.support.design.widget.AppBarLayout>
```

接着重点分析 CollapsingToolbarLayout 布局，简称坍塌（Collapsing）布局。当展开坍塌布局时，显示一张图片，当布局闭合时，显示导航栏，从展开到闭合之间的状态变换，类似于“坍塌”效果，所以称为坍塌布局。坍塌布局包含两个部分，一部分是展开效果的样式，即 PercentFrameLayout，另一部分是闭合效果的样式，即 Toolbar。在坍塌布局内部中，PercentFrameLayout 和 Toolbar 都需要设置 app:layout_collapseMode 属性，以区分在坍塌布局中承担的角色。当属性值为 parallax（视差）时，表示在展开时显示的样式；当属性值为 pin（固定）时，表示在闭合时显示的样式。一般而言，展开样式可以任意选择，如一张图片等，闭合样式一般均为导航栏，即 Toolbar。

下面逐个分析这三个布局样式在开发过程中需要注意的要点。CollapsingToolbarLayout 是整个布局的父布局，管理布局内部的两个子布局，通过 app:contentScrim 属性设置折叠时 Toolbar 布局的主题颜色；通过 app:expandedTitleMarginEnd 和 app:expandedTitleMarginStart 属性设置在

展开状态下标题文字的间距；通过 app:layout_scrollFlags 属性设置滚动标记，scroll 标志表示坍塌布局支持滚动屏幕，未设置布局无法坍塌，将会被固定在屏幕顶部，exitUntilCollapsed 标志表示坍塌布局支持滚动退出屏幕，最后折叠于顶部。PercentFrameLayout 是含有百分比的 FrameLayout 布局，用于在外层切换 ViewPager 时，控制图片动态地淡入或淡出。Toolbar 类似于 ActionBar，是顶部的导航栏，通过 app:popupTheme 属性自定义弹出菜单的样式。

```
<android.support.design.widget.CollapsingToolbarLayout
    android:layout_width="match_parent"
    android:layout_height="400dp"
    android:fitsSystemWindows="true"
    app:contentScrim="?attr/colorPrimary"
    app:expandedTitleMarginEnd="16dp"
    app:expandedTitleMarginStart="16dp"
    app:layout_scrollFlags="scroll|exitUntilCollapsed">

    <android.support.percent.PercentFrameLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:fitsSystemWindows="true"
        app:layout_collapseMode="parallax">

        <ImageView...>

        <ImageView...>

    </android.support.percent.PercentFrameLayout>

    <android.support.v7.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        app:layout_collapseMode="pin"
        app:popupTheme="@style/AppTheme.PopupOverlay"/>

</android.support.design.widget.CollapsingToolbarLayout>
```

在 MainActivity 的 onCreate()方法中，调用 setTitle()方法设置页面的标题，即在 Toolbar 中添加标题。如设置的标题“Girls' Generation”，可以随着导航栏大小的改变，动态地改变字体的大小。

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    // ...

    setTitle("Girls' Generation"); // 设置标题
```


```
}
```

到此，AppBarLayout 布局已经在 app_bar_main.xml 中添加完成，接着编写 AppBarLayout 的控制逻辑，使之可以随着底部 ViewPager 页面切换，而动态地修改顶部图片。

设置 AppBarLayout 布局后的效果如图 2-5 所示。

图 2-5　设置 AppBarLayout 布局后的效果

2.1.4　页面添加 AppBarLayout 逻辑

在 AppBarLayout 布局中，一个重要的部分就是在顶部显示的大幅图片。页面通过图片的形式，传递给用户所要表述的信息，增加对于用户的吸引力，提供更好的用户体验。因此，本例在 AppBarLayout 布局的顶部，使用一组更加复杂的图片样式，在原有展示图片的基础上，再添加一层图片，在图片的切换过程中，模拟渐变的动画效果。继续修改 app_bar_main.xml 布局，在 PercentFrameLayout 布局中，添加两个 ImageView 控件：一个用于展示当前页面的图片，控件的 id 为 appbar_iv_target；一个用于在页面切换时实现页面淡入淡出的动画效果，控件的 id 为 appbar_iv_outgoing。

下面，我们详细分析各个控件的相关属性，即 PercentFrameLayout 布局和两个 ImageView 控件。在原有的 FrameLayout 布局中，PercentFrameLayout 布局添加百分比的属性，子控件支持设置宽高的百分比属性，更加灵活地改变控件的大小。如布局的子控件 ImageView，先把宽度 android:layout_width 属性的值设置为 0，表示控件的宽度根据其他属性动态地计算；再添加宽度百分比 app:layout_widthPercent 属性，属性值设置为“120%”，表示控件的宽度是父控件宽度的 120%，即多延伸 20%的距离，多余部分暂时隐藏。这样设置子控件 ImageView 的宽度，是因为本例所要实现的动画效果既包含左右滑动，又包含淡入淡出，在左右滑动时，多余 20%平均分配每侧 10%，模拟图片的滑入滑出动画。在 PercentFrameLayout 布局中，android:fitsSystemWindows 属性设置为“true”，表示页面填充整个屏幕，包含顶部的状态栏（StatusBar）；app:layout_collapseMode 属性在上文提到过，与坍塌布局相关。ImageView 的属性都是常规属性，android:scaleType 属性设置为“centerCrop”，表示不破坏图片比例的情况下填充整个视图，是非常实用的图片填充属性，app:layout_widthPercent 也在上文提到过，与 PercentFrameLayout 布局相关。

```
<android.support.percent.PercentFrameLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:fitsSystemWindows="true"
    app:layout_collapseMode="parallax">

    <ImageView
        android:id="@+id/appbar_iv_outgoing"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:contentDescription="@null"
        android:scaleType="centerCrop"
        android:visibility="gone"
        app:layout_widthPercent="120%"/>

    <ImageView
        android:id="@+id/appbar_iv_target"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:contentDescription="@null"
        android:scaleType="centerCrop"
        android:src="@drawable/tiffany"
        app:layout_widthPercent="120%"/>

</android.support.percent.PercentFrameLayout>
```

关于获取图片资源的接口，上文有所叙述，放置在 ViewPager 的 Adapter 中，即 SimpleAdapter 的 getDrawable 方法，根据传递的位置信息，获取不同的图片资源文件。

```
public @DrawableRes int getDrawable(int position) {
    if (position >= 0 && position < SECTIONS.length) {
        return SECTIONS[position].getDrawable();
    }
    return -1;
}
```

那么，我们又是如何控制 AppBarLayout 的顶部图片，随着 AppBarLayout 的底部 ViewPager 滑动而变化的呢？在 MainActivity 的 onCreate 方法中，设置 ViewPager 并调用 ViewPager#addOnPageChangeListener()方法，监听 ViewPager 的状态修改，其核心逻辑就是位于 PagerChangeListener 类中。

```
    mVpContainer.addOnPageChangeListener(PagerChangeListener.newInstance(adapter,
mIvTarget, mIvOutgoing));
```

对于监听类 PagerChangeListener，继承 ViewPager 的 OnPageChangeListener 接口，实现接口的三个核心方法，即 onPageScrolled()、onPageScrollStateChanged()、onPageSelected()。onPageScrolled()负责监听页面的滑动过程，其中含有若干参数，position 参数表示滑动开始的位置，positionOffset 参数表示滑动偏移的距离，positionOffsetPixels 参数表示滑动偏移的像素。onPageScrollStateChanged 负责监听页面的滑动状态，其中 state 参数表示滑动状态；onPageSelected()负责监听页面的滑动终止，其中 position 参数表示滑动结束的停留位置。除此之外，PagerChangeListener 类还持有动画管理器类 ImageAnimator，使用 ViewPager 的适配器、两个控制淡入淡出效果的 ImageView，对 ImageAnimator 类进行初始化。

本例主要监听 onPageScrolled()，核心逻辑也均在其中。在 onPageScrolled()方法中，主要使用当前位置参数 position 和滑动偏移参数 positionOffset。

1）首先，调用 isFinishedScrolling()方法，判断动画是否完成，如果正在执行动画，则调用 finishScroll()终止动画，防止动画效果重叠。

2）其次，调用 isStartingScrollToPrevious()方法和 isStartingScrollToNext()方法，判断页面是移动到下一个位置还是前一个位置，根据移动的位置对动画的两个图片控件 ImageView 初始化。

3）最后，调用 isScrollingToNext()方法和 isScrollingToPrevious()方法，判断页面是向前移动还是向后移动，根据移动的偏移执行动画管理器 ImageAnimator 对应的动画效果，ImageAnimator#forward()或 mImageAnimator#backwards()。

读者在这里可能有一个困惑，就是位置变化和移动偏移有什么区别？比如移动到下一个页面（isStartingScrollToNext），在移动过程中，可能会一直移动，那么移动偏移也是到下一个页面（isScrollingToNext），也可能会放弃移动返回当前页面，那么移动偏移就是到前一个页面

（isScrollingToPrevious），同理移动到前一个页面（isStartingScrollToPrevious），也有两种情况：位置变化导致淡入淡出的图片不同；移动偏移导致淡入淡出的动画不同。

```
    public class PagerChangeListener implements ViewPager.OnPageChangeListener {
        private ImageAnimator mImageAnimator; // 图片动画器

        private int mCurrentPosition; // 当前位置
        private int mFinalPosition; // 最终位置
        private boolean mIsScrolling = false; // 是否仍在滑动

        private PagerChangeListener(ImageAnimator imageAnimator) {
            mImageAnimator = imageAnimator;
        }

        public static PagerChangeListener newInstance(SimpleAdapter adapter,
ImageView originImage, ImageView outgoingImage) {
            ImageAnimator imageAnimator = new ImageAnimator(adapter, originImage,
outgoingImage);
            return new PagerChangeListener(imageAnimator);
        }

        @Override
        public void onPageScrolled(int position, float positionOffset, int
positionOffsetPixels) {
            Log.e("DEBUG-WCL", "position: " + position + ", positionOffset: " +
positionOffset);

            // 以前滑动，现在终止
            if (isFinishedScrolling(position, positionOffset)) {
                finishScroll(position);
            }

            // 判断前后滑动是否开始
            if (isStartingScrollToPrevious(position, positionOffset)) {
                startScroll(position);
            } else if (isStartingScrollToNext(position, positionOffset)) {
                startScroll(position + 1); // 向后滚动需要加 1
            }

            // 向后滚动
            if (isScrollingToNext(position, positionOffset)) {
                mImageAnimator.forward(positionOffset);
            } else if (isScrollingToPrevious(position, positionOffset)) {
            // 向前滚动
                mImageAnimator.backwards(positionOffset);
            }
        }
```

```
        @Override
        public void onPageScrollStateChanged(int state) {
            //NO-OP
        }

        @Override
        public void onPageSelected(int position) {
            //NO-OP
        }

        // ...
    }
```

监听类 PagerChangeListener 的其他方法的具体实现细节，这些方法在 onPageScrolled()方法中被调用，相互配合，当页面切换时，执行顶部图片随之切换的动画效果。

```
    // 终止滑动，滑动 && [偏移是 0&&滑动终点] || 动画之中
    private boolean isFinishedScrolling(int position, float positionOffset) {
        return mIsScrolling && (positionOffset == 0f && position == mFinalPosition)
|| !mImageAnimator.isWithin(position);
    }

    // 从静止到开始滑动，下一个，未滑动 && 位置是当前位置 && 偏移量不是 0
    private boolean isStartingScrollToNext(int position, float positionOffset) {
        return !mIsScrolling && position == mCurrentPosition && positionOffset != 0f;
    }

    // 从静止到开始滑动，前一个[position-1]
    private boolean isStartingScrollToPrevious(int position, float positionOffset)
{
        return !mIsScrolling && position != mCurrentPosition && positionOffset != 0f;
    }

    // 开始滚动，向后
    private boolean isScrollingToNext(int position, float positionOffset) {
        return mIsScrolling && position == mCurrentPosition && positionOffset != 0f;
    }

    // 开始滚动，向前
    private boolean isScrollingToPrevious(int position, float positionOffset) {
        return mIsScrolling && position != mCurrentPosition && positionOffset != 0f;
    }

   // 开始滑动，滚动开始，结束位置是position[前滚时position会自动减一]，动画从当前位置到结
      束位置
    private void startScroll(int position) {
        mIsScrolling = true;
        mFinalPosition = position;
```

```
        // 开始滚动动画
        mImageAnimator.start(mCurrentPosition, position);
    }

    // 如果正在滚动，结束时，固定 position 位置，停止滚动，调动截止动画
    private void finishScroll(int position) {
        if (mIsScrolling) {
            mCurrentPosition = position;
            mIsScrolling = false;
            mImageAnimator.end(position);
        }
    }
```

2.1.5　页面添加 AppBarLayout 动画

在 ViewPager 监听类 PagerChangeListener 中，最核心的成员变量就是动画管理类 ImageAnimator，也是本例中非常有趣的部分。ImageAnimator 类实现两种动画的融合，既有淡入淡出效果，又有左右偏移效果，以淡入淡出效果为主，左右偏移效果仅仅偏移部分图片。

在 ImageAnimator 类中，外部成员变量有三个：mAdapter、mTargetImage、mOutgoingImage。其中适配器 mAdapter 提供 mTargetImage 和 mOutgoingImage 的填充图片资源，一个是当前的显示图片，一个是移动的目标图片，当前的显示图片在图片控件 mTargetImage 中设置，移动的目标图片在图片控件 mOutgoingImage 中设置。内部成员变量有四个：FACTOR、mStartPosition、mMinPos、mMaxPos。其中静态常量 FACTOR 表示左右偏移动画的偏移度，部分偏移且偏移范围在 10% 以内；mStartPosition 表示页面移动前的位置信息，用于更新页面图片；mMaxPos 和 mMinPos 分别用于存放开始位置与结束位置之间的最大值和最小值，判断新的输入位置是否在其中。

```
public class ImageAnimator {

    private static final float FACTOR = 0.1f; // 偏移度

    private final SimpleAdapter mAdapter; // 适配器
    private final ImageView mTargetImage; // 原始图片的控件
    private final ImageView mOutgoingImage; // 渐变图片的控件

    private int mStartPosition; // 实际起始位置

    private int mMinPos; // 最小位置
    private int mMaxPos; // 最大位置

    public ImageAnimator(SimpleAdapter adapter, ImageView targetImage, ImageView
```

```
outgoingImage) {
            mAdapter = adapter;
            mTargetImage = targetImage;
            mOutgoingImage = outgoingImage;
        }

        // ...
    }
```

ImageAnimator 类的核心方法主要有两组，一组用于管理动画的启动和结束，一组用于管理向前移动和向后移动的动画效果。动画的启动和结束的方法分别是 start()和 end()方法。start()方法的参数是起始位置 startPosition 和终止位置 endPosition，具体逻辑：

1）储存起始位置 startPosition，用于在滑动结束时通过位置判断是否滑动至下一个页面，还是仍停留在当前页面。

2）根据结束位置 endPosition 获取滑动至下一个页面的图片资源，用于填充目标图片控件 mTargetImage。

3）设置动画图片控件 mOutgoingImage，设置原图片控件的图片资源，初始化偏移，初始化可视性，初始化透明度。

4）使用新的目标图片填充目标图片控件 mTargetImage。

5）储存起始位置 startPosition 和结束位置 endPosition 之间的最小值和最大值，至 mMinPos 和 mMaxPos 中。

```
    /**
     * 启动动画，之后选择向前或向后滑动
     *
     * @param startPosition 起始位置
     * @param endPosition   终止位置
     */
    public void start(int startPosition, int endPosition) {
        mStartPosition = startPosition;

        // 目标图片资源
        @DrawableRes int incomeId = mAdapter.getDrawable(endPosition);

        // 设置动画处理的启示图片
        mOutgoingImage.setImageDrawable(mTargetImage.getDrawable()); // 原始的图片
        mOutgoingImage.setTranslationX(0f);
        mOutgoingImage.setVisibility(View.VISIBLE);
        mOutgoingImage.setAlpha(1.0f);

        // 设置滑动结束的目标图片
        mTargetImage.setImageResource(incomeId);
```

```
        mMinPos = Math.min(startPosition, endPosition); // 最小位置
        mMaxPos = Math.max(startPosition, endPosition); // 最大位置
    }
```

end()方法的参数只有终止位置 endPosition，因为滑动结束，只关心最后的滑动位置，具体逻辑：

1）根据结束位置 endPosition，获取最终页面的图片资源，用于填充目标图片控件 mTargetImage。

2）判断真正的结束位置 endPosition 是否与已保存的起始位置 mStartPosition 相同。如果相同，则表示未滑动至下一个页面，仍停留当前页面，目标图片控件 mTargetImage 继续显示原页面的图片；如果不同，则表示滑动至下一页面，目标图片控件 mTargetImage 设置最终页面的图片资源，设置为非透明，隐藏动画图片控件 mOutgoingImage。

```
    /**
     * 滑动结束的动画效果
     *
     * @param endPosition 滑动位置
     */
    public void end(int endPosition) {
        @DrawableRes int incomeId = mAdapter.getDrawable(endPosition);
        mTargetImage.setTranslationX(0f);

        // 滑动并未执行完成，重新返回当前页面
        if (endPosition == mStartPosition) {
            mTargetImage.setImageDrawable(mOutgoingImage.getDrawable());
        } else { // 滑动执行完成，隐藏动画图片
            mTargetImage.setImageResource(incomeId);
            mTargetImage.setAlpha(1f);
            mOutgoingImage.setVisibility(View.GONE);
        }
    }
```

在 ImageAnimator 类中，另一组用于管理向前移动和向后移动的动画效果的核心方法是 forward()和 backwards()。forward()方法和 backwards()方法的参数均是位置偏移 positionOffset，根据偏移量设置目标图片控件 mTargetImage 和动画图片控件 mOutgoingImage 的水平偏移度（TranslationX）和透明度（Alpha），水平偏移的阈值通过图片宽度和 FACTOR 参数控制。向前移动和向后移动的动画效果逻辑相似，数值相反。

```
    /**
     * 向前滚动，比如 0->1，offset 滚动的距离(0->1)，目标渐渐淡出
     *
     * @param positionOffset 位置偏移
     */
    public void forward(float positionOffset) {
```

```
        Log.e("DEBUG-WCL", "forward-positionOffset: " + positionOffset);
        int width = mTargetImage.getWidth();
        mOutgoingImage.setTranslationX(-positionOffset * (FACTOR * width));
        mTargetImage.setTranslationX((1 - positionOffset) * (FACTOR * width));

        mTargetImage.setAlpha(positionOffset);
    }

    /**
     * 向后滚动，比如 1->0，offset 滚动的距离(1->0)，目标渐渐淡入
     *
     * @param positionOffset 位置偏移
     */
    public void backwards(float positionOffset) {
        Log.e("DEBUG-WCL", "backwards-positionOffset: " + positionOffset);
        int width = mTargetImage.getWidth();
        mOutgoingImage.setTranslationX((1 - positionOffset) * (FACTOR * width));
        mTargetImage.setTranslationX(-(positionOffset) * (FACTOR * width));

        mTargetImage.setAlpha(1 - positionOffset);
    }
```

最后一个方法是 isWithin()，用于判断位置 position 是否在最小位置 mMinPos 与最大位置 mMaxPos 之间，用于判断动画的执行界限，防止快速滑动导致动画重叠。

```
    // 判断位置是否在其中，用于停止动画
    public boolean isWithin(int position) {
        return position >= mMinPos && position < mMaxPos;
    }
```

AppBarLayout 布局的收起和展开如图 2-6 所示。

至此为止，关于 Material Design 中的重要控件 AppBarLayout 布局，其全部的开发知识要点，已经通过实例完整地讲述，比起空白的知识罗列，在实例中边做边学，可以达到更好的学习效果。本节的知识点包括以下内容：

1）搭建项目框架。使用一个新颖的 Navigation Drawer Activity（导航抽屉）页面作为项目的主页面，在其中设置若干主题颜色和修改抽屉显示样式，为添加 AppBarLayout 布局做准备。

2）页面设置 ViewPager 布局。ViewPager 布局是与 AppBarLayout 布局联动的页面，根据 ViewPager 布局的页面切换位置，动态地改变 AppBarLayout 布局的显示图片，在 ViewPager 的监听接口中，实现联动逻辑。

3）页面添加 AppBarLayout 布局。AppBarLayout 布局作为 MD 中重要的控件，支持与大量控件组合使用，如本例中的 CoordinatorLayout、NestedScrollView、CollapsingToolbarLayout、TabLayout、PercentFrameLayout、Toolbar 等。本例详细地分析了如何使用和组合这些组件，使

每个组件发挥各自的作用，构成一个与 AppBarLayout 布局关联的整体。

图 2-6　AppBarLayout 布局的收起和展开

4）页面添加 AppBarLayout 逻辑。AppBarLayout 布局的逻辑部分，是通过与 ViewPager 布局的联动进行分析的。本例详细地分析了在 ViewPager 中 OnPageChangeListener 监听是如何关联 AppBarLayout 布局与 ViewPager 布局，执行动画的切换效果的。

5）页面添加 AppBarLayout 动画。这是一个有趣的部分，关于动画效果，本例通过巧妙地设计 ImageAnimator 类，改变图片内容，通过左右平移与淡入淡出相结合，给予用户更加精致的体验。

对于高级开发者而言，熟练地使用系统提供的各种高阶基础控件是必备技能，同时，还要掌握控件之间的关联，以做到游刃有余，减少因知识不足所导致地重复轮子开发。

2.2　熟练掌握 CoordinatorLayout 的开发技术

CoordinatorLayout，顾名思义为协调布局，属于功能更加强大的 FrameLayout，适用于两种使用方式：第一，作为页面最顶层的装饰（Decor）布局；第二，作为包含一个或多个子视图的特定交互容器。CoordinatorLayout 通过为子控件设置交互行为（Behavior），支持与多个子视图进行交互，同时，也支持子视图之间进行相互交互。当控件作为 CoordinatorLayout 的子控件时，在控件的类中，方法通过设置 DefaultBehavior 注解提供默认的交互行为。在 CoordinatorLayout

中，行为可用于实现在布局中各种子控件的交互与附着，如滑动抽屉或面板、滑动页面控件或按钮，这些控件伴随着动画效果粘附于其他控件之上。

CoordinatorLayout 作为 Android 的高级控件受到重点推荐，曾经取代 LinearLayout，作为默认 HelloWorld 项目的顶层控件；同时，也是 Material 风格的重要组件，用于协调内部的其他控件，实现相互联动。下面来看看 CoordinatorLayout 如何使用吧！

希望通过本节让读者掌握 CoordinatorLayout 的实现原理，不仅知道如何使用，还要知道何时使用，根据项目的需求，实现个性化定制。

本文实例完整代码的下载地址为 ttps://github.com/SpikeKing/TestCoordinatorLayout.git。

2.2.1 项目框架

首先创建默认工程——HelloWorld 项目即可，导入三个第三方开源的依赖库。circleimageview 库用于显示圆形头像；cardview 库用于提供卡片样式的展示视图；butterknife 库用于在源码中绑定页面布局的 ID，可以使用最近版本。

```
compile 'de.hdodenhof:circleimageview:1.3.0'
compile 'com.android.support:cardview-v7:23.1.0'
compile 'com.jakewharton:butterknife:7.0.1'
```

本实例以 CoordinatorLayout 为核心，主要布局位于 activity_main.xml 中，主要逻辑位于 MainActivity 中。

2.2.2 布局设计

为了更好地展示 CoordinatorLayout 的协调控件特性，本例以 CoordinatorLayout 为根布局，其内部含有四个部分，即 AppBarLayout、NestedScrollView、Toolbar、CircleImageView。AppBarLayout 用于展示顶部的图片视图，以及缩放联动；NestedScrollView 用于展示中部的文本内容，含有滚动功能；Toolbar 用于展示顶部的动作条；CircleImageView 用于展示圆形头像。CoordinatorLayout 用于管理这四个部分的联动，联动既包含与父控件（CoordinatorLayout）的联动，也包含相互之间的联动。CoordinatorLayout 的逻辑实现位于 MainActivity 中。

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    tools:context="org.wangchenlong.testcoordinatorlayout.MainActivity">
    <android.support.design.widget.AppBarLayout/>
    <android.support.v4.widget.NestedScrollView/>
    <android.support.v7.widget.Toolbar/>
    <de.hdodenhof.circleimageview.CircleImageView/>
```

```
</android.support.design.widget.CoordinatorLayout>
```

第一个，AppBarLayout 是顶层的图片展示布局，通过设置内部布局的 layout_scrollFlags 属性，为其提供动画支持。在 AppBarLayout 的内部，含有 CollapsingToolbarLayout，通过坍塌的动画效果转变为 Toolbar，设置属性 layout_scrollFlags，scroll 表示随着滚动布局坍塌，snap 表示子视图向上滚动具有吸附效果。在 CollapsingToolbarLayout 的内部，含有 ImageView 和 FrameLayout，一个是图片视图，一个是帧布局，图片视图用于显示顶部图片，帧布局用于显示底部内容，含有若干文字信息。两者都设置 layout_collapseMode 属性为 parallax，具有视差效果，随着滑动一起压缩直至顶部；如果设置为 pin，则位置固定，不能随着滑动而移动。

```
<android.support.design.widget.AppBarLayout
    android:id="@+id/main_abl_app_bar">
    <android.support.design.widget.CollapsingToolbarLayout
        android:layout_height="450dp"
        app:layout_scrollFlags="scroll|snap">
        <ImageView
            android:id="@+id/main_iv_placeholder"
            android:layout_height="300dp"
            android:scaleType="centerCrop"
            android:src="@drawable/large"
            app:layout_collapseMode="parallax" />
        <FrameLayout
            android:id="@+id/main_fl_title"
            android:layout_height="150dp"
            android:layout_gravity="bottom|center_horizontal"
            android:background="@color/colorPrimary"
            app:layout_collapseMode="parallax">
        </FrameLayout>
    </android.support.design.widget.CollapsingToolbarLayout>
</android.support.design.widget.AppBarLayout>
```

在 FrameLayout 中含有 LinearLayout，用于显示页面的标题信息，使用两层 Layout 嵌套的原因是内层随着页面滑动，动态地显示或者消失。

```
<LinearLayout
    android:id="@+id/main_ll_title_container"
    android:orientation="vertical">
    <TextView
        android:layout_marginTop="@dimen/title_margin"
        android:layout_gravity="center_horizontal"
        android:gravity="bottom|center"
        android:text="@string/person_name"/>
    <TextView
        android:layout_gravity="center_horizontal"
        android:layout_marginTop="4dp"
        android:text="@string/person_title"/>
</LinearLayout>
```

注意，内层有些属性的实现依赖于外层布局的逻辑，如 CollapsingToolbarLayout 的 layout_scrollFlags 属性，是由 AppBarLayout 实现效果；ImageView 和 FrameLayout 的 layout_collapseMode 属性是由 CollapsingToolbarLayout 实现效果。最终的样式如图 2-7 所示。

图 2-7　AppBarLayout 最终的样式

第二个，NestedScrollView 是滚动视图，内部填充需要滚动的视图。注意视图 View 和布局 Layout 之间的差别，根据 Android 的命名规范，View 的内部只支持嵌套一个控件，用于扩展控件的样式，而 Layout 的内部支持嵌套多个控件，管理这些控件的行为。NestedScrollView 的 behavior_overlapTop 属性表示顶部重叠的距离，layout_behavior 属性表示特殊行为，如值 appbar_scrolling_view_behavior 是当前视图随着 AppBar 的滚动而向上滑动。在 NestedScrollView 中是一个卡片视图 CardView，cardElevation 属性表示阴影深度，contentPadding 属性表示内容到边框的距离。在 CardView 中是一个文字视图 TextView，lineSpacingExtra 属性表示文字每行的间距。

```
<android.support.v4.widget.NestedScrollView
    android:scrollbars="none"
    app:behavior_overlapTop="30dp"
    app:layout_behavior="@string/appbar_scrolling_view_behavior">
    <android.support.v7.widget.CardView
        app:cardElevation="8dp"
        app:contentPadding="16dp">
        <TextView
            android:lineSpacingExtra="8dp"
            android:text="@string/person_intro" />
    </android.support.v7.widget.CardView>
</android.support.v4.widget.NestedScrollView>
```

最终是带滚动（NestedScrollView）的卡片样式（CardView）文字（TextView）视图，如图 2-8 所示。

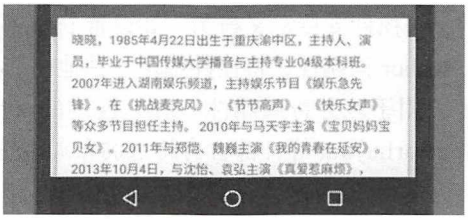


图 2-8　带滚动的卡片样式文字视图

第三个，Toolbar 是动作条，高度 layout_height 被设置为 ActionBar 的默认高度?attr/actionBarSize，推荐使用 Toolbar 替代默认的 ActionBar，与页面其他元素实现互动，定制更多样式。Toolbar 的 layout_anchor 表示锚点位置，将锚点放置在 CollapsingToolbarLayout 的 FrameLayout 布局中，则 FrameLayout 的上部被 Toolbar 占据，逻辑位于顶层布局 CoordinatorLayout 中，同时将 Toolbar 的颜色与锚点布局的颜色设为一致，隐藏于其中。Toolbar 的内部是一个水平布局的 LinearLayout，含有一个占位的 Space 控件，用于之后替换图片控件，还含有一个 TextView 用于显示标题，Space 控件默认无法显示，TextView 的显示状态（visibility）设置为隐藏。这样做，可将 Toolbar 完美地隐藏于锚点布局 FrameLayout 中。

```
<android.support.v7.widget.Toolbar
    android:id="@+id/main_tb_toolbar"
    android:layout_height="?attr/actionBarSize"
    android:background="@color/colorPrimary"
    app:layout_anchor="@id/main_fl_title">
    <LinearLayout
        android:orientation="horizontal">
        <Space
            android:layout_width="@dimen/image_final_width"
            android:layout_height="@dimen/image_final_width" />
        <TextView
            android:id="@+id/main_tv_toolbar_title"
            android:text="@string/person_name"
            android:textColor="@android:color/white"
            android:visibility="invisible"/>
    </LinearLayout>
</android.support.v7.widget.Toolbar>
```

第四个比较简单，开源的圆形图片控件，border_color 属性表示图片控件的边框颜色，border_width 属性表示图片控件的边框宽度，layout_behavior 表示自定义行为 AvatarImageBehavior，根布局 CoordinatorLayout 统一管理行为。

```
<de.hdodenhof.circleimageview.CircleImageView
    android:src="@drawable/small"
    app:border_color="@android:color/white"
    app:border_width="2dp"
    app:layout_behavior=".AvatarImageBehavior" />
```


在 CoordinatorLayout 中，子控件支持定义行为，如设置 layout_behavior 行为属性，实现特定的动画效果，设置 layout_anchor 锚属性，将控件锚定于其他内部控件中。以 layout 起始属性表示逻辑控件位于父布局中，用于区分普通属性。MD 样式的布局，如 CoordinatorLayout、AppBarLayout、CollapsingToolbarLayout 等，都含有特定的布局属性供子控件调用。

2.2.3 联动逻辑

核心逻辑位于 MainActivity 中，以 onCreate()为起点。为 Activity 设置布局的 setContentView 函数，将 activity_main 布局关联至 MainActivity；ButterKnife 将 Activity 自动绑定 activity_main 布局中的属性；调用 mTbToolbar#setTitle()设置页面顶部的标题为空。为 AppBarLayout（即 mAblAppBar）添加偏移（Offset）监听，即调用 addOnOffsetChangedListener()，当 AppBarLayout 向上或向下滑动时，会触发监听，调用匿名类 OnOffsetChangedListener 的 onOffsetChanged()函数，执行相应的逻辑。调用 initParallaxValues()设置自动滑动效果，即视差，可以让 View 一起滚动，并控制滚动视觉差。进而将逐个解析 AppBarLayout 的联动逻辑。

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ButterKnife.bind(this);
        mTbToolbar.setTitle("");

        // 向上或向下滑动 AppBar 会触发监听
        mAblAppBar.addOnOffsetChangedListener(new
AppBarLayout.OnOffsetChangedListener() {
            @Override
            public    void    onOffsetChanged(AppBarLayout    appBarLayout,    int
verticalOffset) {
                int maxScroll = appBarLayout.getTotalScrollRange();
                float  percentage  =  (float)  Math.abs(verticalOffset)  /  (float)
maxScroll;
                handleAlphaOnTitle(percentage);
                handleToolbarTitleVisibility(percentage);
            }
        });

        initParallaxValues(); // 设置自动滑动（视差）效果
    }
```

监听 AppBarLayout 的偏移距离，调用 AppBarLayout#getTotalScrollRange()，获取滚动的最大距离，计算滚动的百分比，调用 handleAlphaOnTitle()和 handleToolbarTitleVisibility()，根据百分比做出相应的反馈，控制中心的标题和顶部的标题显示。

```
    mAblAppBar.addOnOffsetChangedListener(new
AppBarLayout.OnOffsetChangedListener() {
        @Override
        public void onOffsetChanged(AppBarLayout appBarLayout, int verticalOffset)
{
            int maxScroll = appBarLayout.getTotalScrollRange();
            float percentage = (float) Math.abs(verticalOffset) / (float) maxScroll;
            handleAlphaOnTitle(percentage);
            handleToolbarTitleVisibility(percentage);
        }
    });
```

函数 handleAlphaOnTitle()控制中心位置的标题（Title）显示，默认显示中心的标题。当百分比大于 PERCENTAGE_TO_HIDE_TITLE_DETAILS（30%）时，如果标题布局状态（mIsTheTitleContainerVisible）可视，则调用透明度动画，将标题设置为非透明，将标题布局状态设置为不可视；如果标题布局状态不可视，则将标题设置为透明，将标题布局状态设置为可视。透明度动画的执行时间，即 AlphaAnimation#setDuration()，设置为 ALPHA_ANIMATIONS_DURATION（200 毫秒）；当透明度动画执行完成后，保持最终状态，即 AlphaAnimation#setFillAfter(true)。

```
    private void handleAlphaOnTitle(float percentage) {
        if (percentage >= PERCENTAGE_TO_HIDE_TITLE_DETAILS) {
            if (mIsTheTitleContainerVisible) {
                startAlphaAnimation(mLlTitleContainer,  ALPHA_ANIMATIONS_DURATION,
View.INVISIBLE);
                mIsTheTitleContainerVisible = false;
            }
        } else {
            if (!mIsTheTitleContainerVisible) {
                startAlphaAnimation(mLlTitleContainer,  ALPHA_ANIMATIONS_DURATION,
View.VISIBLE);
                mIsTheTitleContainerVisible = true;
            }
        }
    }

    public static void startAlphaAnimation(View v, long duration, int visibility)
{
        AlphaAnimation alphaAnimation = (visibility == View.VISIBLE)
                ? new AlphaAnimation(0f, 1f)
                : new AlphaAnimation(1f, 0f);

        alphaAnimation.setDuration(duration);
        alphaAnimation.setFillAfter(true);
        v.startAnimation(alphaAnimation);
    }
```


函数 handleToolbarTitleVisibility()控制顶部 ToolBar 的标题显示，默认不显示 ToolBar 的标题，ToolBar 隐藏于中心部位。当百分比大于 PERCENTAGE_TO_SHOW_TITLE_AT_TOOLBAR（90%）时，通过 mIsTheTitleVisible 状态，调用透明度动画，控制 ToolBar 的标题显示，显示逻辑类似于 handleAlphaOnTitle()。

```
    private void handleToolbarTitleVisibility(float percentage) {
        if (percentage >= PERCENTAGE_TO_SHOW_TITLE_AT_TOOLBAR) {
            if (!mIsTheTitleVisible) {
                startAlphaAnimation(mTvToolbarTitle,     ALPHA_ANIMATIONS_DURATION,
View.VISIBLE);
                mIsTheTitleVisible = true;
            }
        } else {
            if (mIsTheTitleVisible) {
                startAlphaAnimation(mTvToolbarTitle,     ALPHA_ANIMATIONS_DURATION,
View.INVISIBLE);
                mIsTheTitleVisible = false;
            }
        }
    }
```

中心标题与顶部 ToolBar 标题的显示位置如图 2-9 所示。

图 2-9　中心标题与顶部 ToolBar 标题的显示位置

函数 initParallaxValues()设置视差动画的显示效果，设置相应的滑动比率，在布局中设置 CollapsingToolbarLayout 的 layout_collapseMode 属性为 parallax，即视差动画模式。获取顶部图片 mIvPlaceholder 的布局参数（LayoutParams），获取中部标题 mFlTitleContainer 的布局参数，将顶部图片的视差值设置为 0.9，将中部标题的视差值设置为 0.3，并且覆盖已有的布局参数。视差值的含义是指不会随着布局滑动的比例，如果设置为 1，则 View 不会滚动；如果设置为 0，则 View 就会同步滚动；如果设置为 0~1 的中间值，则表示不随滚动的比例。

```
    private void initParallaxValues() {
        CollapsingToolbarLayout.LayoutParams petDetailsLp =
                (CollapsingToolbarLayout.LayoutParams)
mIvPlaceholder.getLayoutParams();

        CollapsingToolbarLayout.LayoutParams petBackgroundLp =
                (CollapsingToolbarLayout.LayoutParams)
mFlTitleContainer.getLayoutParams();
        petDetailsLp.setParallaxMultiplier(0.9f);
        petBackgroundLp.setParallaxMultiplier(0.3f);

        mIvPlaceholder.setLayoutParams(petDetailsLp);
        mFlTitleContainer.setLayoutParams(petBackgroundLp);
    }
```

联动布局的逻辑已经设置完成，核心是随着 AppBarLayout 的滚动，修改中部和顶部标题的透明度，达到隐藏或显示标题的目的。同时，设置视差（Parallax）控制滚动的比例。

2.2.4　图片交互

本例含有一个自定义的交互逻辑 AvatarImageBehavior，控制中心图片随着 ToolBar 的滑动，产生位置和尺寸的变化。在布局中，CircleImageView 控件设置 layout_behavior 属性，交互类为 AvatarImageBehavior。

```
    <de.hdodenhof.circleimageview.CircleImageView
        android:layout_width="@dimen/image_width"
        android:layout_height="@dimen/image_width"
        android:layout_gravity="center"
        android:src="@drawable/small"
        app:border_color="@android:color/white"
        app:border_width="2dp"
        app:layout_behavior=".AvatarImageBehavior"/>
```

AvatarImageBehavior 继承于 CoordinatorLayout.Behavior 类，类参数为 CircleImageView，即交互类 AvatarImageBehavior 所应用的控件。

```
    public class AvatarImageBehavior extends CoordinatorLayout.Behavior
<CircleImageView>
```

覆写接口 layoutDependsOn()，表示交互行为的作用控件（CircleImageView）与 Toolbar 控件（依赖控件）联动。

```
    @Override
    public boolean layoutDependsOn(CoordinatorLayout parent, CircleImageView child,
View dependency) {
        return dependency instanceof Toolbar;  // 依赖 Toolbar 控件
    }
```

互动的核心逻辑位于 onDependentViewChanged()函数，根据作用控件 CircleImageView 与依赖控件 dependency 初始化属性，根据当前高度与最大高度，计算出已经滑动的百分比 expandedPercentageFactor；根据原始位置和最终位置的差值，结合百分比，计算出水平和竖直的滑动距离，还有高度的滑动距离。最后，根据三个滑动距离设置 CircleImageView 的水平位置和竖直位置，还有大小。

```
    @Override
    public boolean onDependentViewChanged(CoordinatorLayout parent,
CircleImageView child, View dependency) {
        shouldInitProperties(child, dependency);  // 初始化属性
        final int maxScrollDistance = (int) (mStartToolbarPosition -
getStatusBarHeight());  // 最大滑动距离：起始位置-状态栏高度
        float expandedPercentageFactor = dependency.getY() / maxScrollDistance;
        // 滑动的百分比

        float distanceYToSubtract = ((mStartYPosition - mFinalYPosition)
                * (1f - expandedPercentageFactor)) + (child.getHeight() / 2);
        // Y 轴距离
        float distanceXToSubtract = ((mStartXPosition - mFinalXPosition)
                * (1f - expandedPercentageFactor)) + (child.getWidth() / 2);
        // X 轴距离

        float heightToSubtract = ((mStartHeight - mFinalHeight) * (1f -
expandedPercentageFactor));
        // 高度减小

        // 设置图片位置
        child.setY(mStartYPosition - distanceYToSubtract);
        child.setX(mStartXPosition - distanceXToSubtract);

        // 设置图片大小
        CoordinatorLayout.LayoutParams lp = (CoordinatorLayout.LayoutParams)
child.getLayoutParams();
        lp.width = (int) (mStartHeight - heightToSubtract);
        lp.height = (int) (mStartHeight - heightToSubtract);
```


```
        child.setLayoutParams(lp);

        return true;
    }
```

在 shouldInitProperties()函数中，根据作用控件 CircleImageView 和依赖控件计算出作用控件 CircleImageView 的水平位置、竖直位置、控件大小的起始和最终值。同时，计算出 ToolBar 的起始位置。

```
    private void shouldInitProperties(CircleImageView child, View dependency) {
        if (mStartXPosition == 0)  // 图片控件水平中心
            mStartXPosition = (int) (child.getX() + (child.getWidth() / 2));
        if (mFinalXPosition == 0)  // 边缘+缩略图宽度的一半
            mFinalXPosition  =  mContext.getResources().getDimensionPixelOffset(R.
dimen.abc_action_bar_content_inset_material) + (mFinalHeight / 2);

        if (mStartYPosition == 0)  // 图片控件竖直中心
            mStartYPosition = (int) (child.getY() + (child.getHeight() / 2));
        if (mFinalYPosition == 0)  // Toolbar 中心
            mFinalYPosition = (dependency.getHeight() / 2);

        if (mStartHeight == 0)  // 图片高度
            mStartHeight = child.getHeight();
        if (mFinalHeight == 0)  // Toolbar 缩略图高度
            mFinalHeight = mContext.getResources().getDimensionPixelOffset(R.dimen.
image_final_width);

        if (mStartToolbarPosition == 0)  // Toolbar 的起始位置
            mStartToolbarPosition = dependency.getY() + (dependency.getHeight() / 2);
    }
```

至此，当 ToolBar 由中部位置滑动至顶部位置时，中心的图片 CircleImageView 也随之滑动，并逐步缩小，反之亦然，效果如图 2-10 所示。

在 CoordinatorLayout 布局中，通过设置交互行为，即 layout_behavior 属性，实现子控件与父控件之间的联动，或者不同子控件之间的联动。以 layout_为开始的属性，均与父控件相关，父控件控制子控件的变化效果，滑动（layout_scrollFlags）或者坍塌（layout_collapseMode）。

◎ 在本例中，顶层控件 CoordinatorLayout 控制 AppBarLayout、NestedScrollView、Toolbar 和 CircleImageView 四个组件，通过设置联动属性，实现控件之间的互动。NestedScrollView、Toolbar 和 AppBarLayout 同步滚动；CircleImageView 随着 Toolbar 的变化，产生位移和尺寸的变化。

◎ 在本例中，通过两种不同的方式，实现控件的形态变化，标题控件由中部到顶部，通过透明度的变化，中部控件逐渐消失，顶部控件逐渐显示；图片控件（CircleImageView）由中部到顶部，则通过交互动画绑定 ToolBar，随之移动位置、

改变尺寸。

图 2-10　中心图片效果

在 CoordinatorLayout 布局中，通过控件之间的互动，创建更多的动画效果，提供更多的联动操作，提升用户体验。CoordinatorLayout 布局作为 Material Design 的核心控件，高级 Android 开发者需要重点关注和熟练掌握，艺无止境，与君共勉。

2.3　熟练掌握 ConstraintLayout 的开发技术

在 Android 开发中，布局设计是一个重要的部分。当页面中含有多组控件，控件之间的关联又十分密切时，如何设计出良好的布局是开发者必须要面对的问题。对于优秀的布局设计而言：

◎　第一点，布局嵌套的层级尽可能的少，当嵌套层级大于 3 层时，可能导致渲染速度过慢，影响加载页面的流畅性，导致页面丢帧。而在极端情况下，甚至会导致 ANR（Application Not Responding）情况的发生，即应用无响应，非常影响用户体验和产品性能。

◎　第二点，布局中控件应该尽可能地使用相对位置，这样在不同分辨率的设备中，仍然能够保持布局效果的一致性。不同分辨率下的宽和高的 DP 总值是不同的，如果使用绝对的 DP 值，则会导致在不同设备下，页面的展示效果不一致，可能发生拉伸或遮挡。因此，适配不同设备也是测试部门必备的测试步骤之一。

但是，这两点往往是冲突的，降低嵌套层级的方式是使用 RelativeLayout，而基于相对位置的方式是使用 LinearLayout。当页面过于复杂时，需要大量的 LinearLayout 保证控件的相对摆放，这样会导致层级过多，如果使用 RelativeLayout 布局，需要将页面设计得非常复杂，控件之间的关系极其冗余，导致页面设计很难理解，也很难扩展，影响布局代码的可读性。这个是一个难题，源于 Android SDK 提供的工具还不够。这就是 ConstraintLayout 需要解决的问题。

ConstraintLayout，即约束性布局，在 2016 年的 Google I/O 大会中推出，从支持力度而言，将会成为主流的布局样式，部分代替原有的布局。在开发布局的过程中，使用 ConstraintLayout 布局可以减少布局的层级，同时优化布局的渲染性能。在一些 Android Studio 的版本中，ConstraintLayout 已经取代了 RelativeLayout，成为 Hello World 工程的默认布局，推荐开发者们使用。不过，ConstraintLayout 并未集成至 SDK 中，而是作为非绑定（Unbundled）的支持库去引用，命名空间是“app:”，即来源于本地包的命名空间。

本文实例完整代码的下载地址为 https://github.com/SpikeKing/ConstraintLayoutDemo。

2.3.1　工程配置

ConstraintLayout 的支持库需要额外导入至工程配置 build.gradle 中。在 SDK Tools 中，位于 Support Repository -> ConstraintLayout for Android，版本号是 1.0.2，如图 2-11 所示。

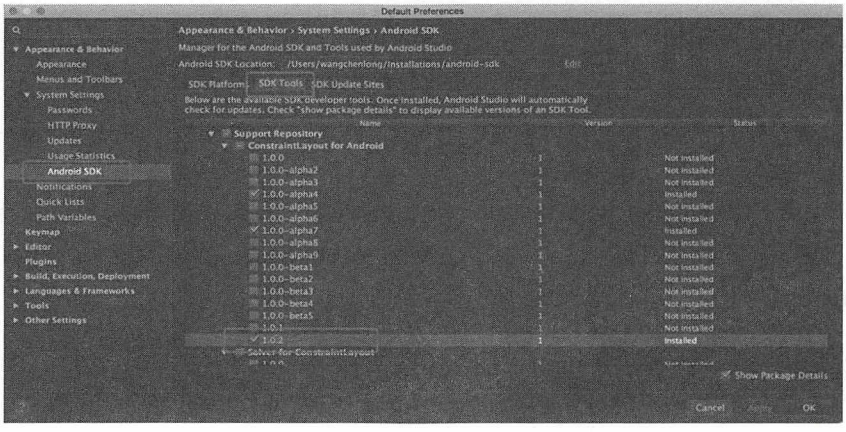


图 2-11　工程配置

在 build.gradle 中，添加 ConstraintLayout 的依赖库 constraint-layout。

```
dependencies {
    compile 'com.android.support.constraint:constraint-layout:1.0.2'
}
```

本例为了展示 ConstraintLayout 的不同特性，创建多个页面样式用于展示，因此，首页 MainActivity 使用列表样式 ListView，将不同页面的入口连接到一起，每一项对应不同的布局页面。

在首页 MainActivity 的入口函数 onCreate()中，首先，页面填充首页布局 activity_main，其中含有一个列表控件 ListView；其次，为 ListView 添加列表适配器 ArrayAdapter；最后，为 ListView 添加列表项点击监听 OnItemClickListener。当点击列表项时，根据点击位置 i，将页面标题 EXTRA_TITLE 和页面布局 EXTRA_LAYOUT_ID 传入待启动页面 LayoutDisplayActivity，每一个页面都显示不同的标题和布局，用于展示 ConstraintLayout 的不同特性。

```
@Override protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);  // 布局

    ListView list = (ListView) findViewById(R.id.activity_main);  // 列表控件
    ArrayAdapter<String> adapter =
            new ArrayAdapter<>(this, android.R.layout.simple_list_item_1, LIST_ ITEMS);
    list.setAdapter(adapter);
    list.setOnItemClickListener(new AdapterView.OnItemClickListener() {
        @Override
        public void onItemClick(AdapterView<?> adapterView, View view, int i, long l) {
            // 复用页面，显示不同的布局样式
            Intent intent = new Intent(MainActivity.this, LayoutDisplayActivity.
class);
            intent.putExtra(Intent.EXTRA_TITLE, LIST_ITEMS[i]); // 标题
            intent.putExtra(LayoutDisplayActivity.EXTRA_LAYOUT_ID,
LAYOUT_IDS[i]); // 布局 ID
            startActivity(intent);
        }
    });
}
```

在展示页面 LayoutDisplayActivity 中，逻辑非常简单，调用 setTitle()将启动参数中的标题 EXTRA_TITLE 设置为页面标题，调用 setContentView()将启动参数中的布局 EXTRA_LAYOUT_ID 设置为页面布局。

```
public class LayoutDisplayActivity extends AppCompatActivity {
    private static final String TAG = LayoutDisplayActivity.class.
getSimpleName();
    static final String EXTRA_LAYOUT_ID = TAG + ".layoutId"; // 布局 ID
```


```
    @Override protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setTitle(getIntent().getStringExtra(Intent.EXTRA_TITLE));
        final int layoutId = getIntent().getIntExtra(EXTRA_LAYOUT_ID, 0);
        setContentView(layoutId); // 设置页面布局，复用布局
    }
}
```

通过不同的展示页面，可以将不同的 ConstraintLayout 特性展示出来。

2.3.2　约束布局

ConstraintLayout 约束布局的含义是：根据布局中的其他元素或视图控件，确定当前视图控件在屏幕中的位置，视图控件受到三类约束，即其他视图控件、父容器（parent）、基准线（Guideline）。其中，ConstraintLayout 约束布局的标准属性模板如下：

```
layout_constraint[本源位置]_[目标位置]="[目标 ID]"
```

例如：

```
app:layout_constraintBottom_toBottomOf="@+id/constraintLayout"
```

即在约束布局 ConstraintLayout 中，视图控件的约束是：将“当前视图的底部”约束至“目标视图的底部”，目标视图的名称 ID 是 constraintLayout。也就是说，把当前视图的底部对齐到 constraintLayout 布局的底部。这是约束布局的表述方式，在约束布局中，所有复杂的排列都是通过类似的形式组合在一起的。

1. 指定约束

在 ConstraintLayout 约束布局中，最基本的使用方式就是指定约束，类似于 RelativeLayout 的排列，但是比其更加灵活。在本例中，点击首页 MainActivity 的 Base 项，即可跳转展示最基本的使用方式页面，即布局 R.layout.layout_base。

在布局 layout_base 中：

◎ “取消按钮”（cancel_button）。

◎ layout_constraintBottom_toBottomOf="@id/constraintLayout：底部（Bottom）对齐“父容器”（constraintLayout）的底部（Bottom）。

◎ layout_constraintStart_toStartOf="@id/constraintLayout：左侧（Start）对齐“父容器”（constraintLayout）的左侧（Start）。

◎ “下一步按钮”（next_button）按钮。

◎ layout_constraintBottom_toBottomOf="@id/constraintLayout：底部（Bottom）对齐“父容器”（constraintLayout）的底部（Bottom）。

◎ layout_constraintStart_toEndOf="@id/cancel_button"：左侧（Start）对齐“取消按钮”（cancel_button）的右侧（End）。

◎ 在每个按钮中，底部（layout_marginBottom）和左侧（layout_marginStart）都会添加空隙间隔（Margin）。

对于 ConstraintLayout 的属性设置，最重要的是理解属性所表达的含义，才能熟练使用。父容器既可以直接指定 ID，如 constraintLayout，也可以使用“parent”代指，效果是相同的。

```
<android.support.constraint.ConstraintLayout
    android:id="@+id/constraintLayout">
    <Button
        android:id="@+id/cancel_button"
        android:layout_marginBottom="16dp"
        android:layout_marginStart="16dp"
        android:text="取消"
        app:layout_constraintBottom_toBottomOf="@id/constraintLayout"
        app:layout_constraintStart_toStartOf="@id/constraintLayout"/>

    <Button
        android:id="@+id/next_button"
        android:layout_marginBottom="16dp"
        android:layout_marginStart="16dp"
        android:text="下一步"
        app:layout_constraintBottom_toBottomOf="@id/constraintLayout"
        app:layout_constraintStart_toEndOf="@id/cancel_button"/>
</android.support.constraint.ConstraintLayout>
```

最终的效果是在页面的左下角水平排列两个按钮，“取消”和“下一步”，如图 2-12 所示。

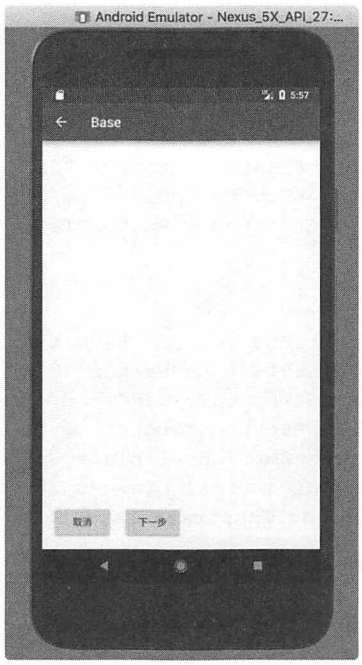


图 2-12　“取消”和“下一步”按钮

2. 设置比例

在 ConstraintLayout 约束布局中，除了指定约束，还支持设置比例，类似于 LinearLayout 中的 weight 属性，但是比其更加灵活。在本例中，单击首页 MainActivity 的 Bias 项，即可跳转展示设置比例的页面，即布局 R.layout.layout_bias。

在布局 layout_bias 中：

◎ “中心按钮”（Center）：上下左右（Top、Bottom、Start、End）均对齐“父控件”（constraintLayout）的上下左右（Top、Bottom、Start、End），即表示“中心按钮”位于页面的中心。

◎ “比例按钮”（Bias）：上下左右也均对齐“父控件”（constraintLayout）的上下左右，同时还设置在页面中水平与竖直的比例。

◎ layout_constraintHorizontal_bias="0.25"：设置水平比例为 0.25，方向是从左至右，左侧比右侧为“0.25 : 1”，即“1 : 4”。

◎ layout_constraintVertical_bias="0. 25"：设置竖直比例为 0.25，方向是从上到下，即上部比下部为“0.25 : 1”，即“1 : 4”。

对于设置控件在父容器中的位置比例，需要提前对齐父容器的四个边，即上下左右。

```
<android.support.constraint.ConstraintLayout
    android:id="@+id/constraintLayout">
    <Button
        android:text="Center"
        app:layout_constraintEnd_toEndOf="@id/constraintLayout"
        app:layout_constraintStart_toStartOf="@id/constraintLayout"
        app:layout_constraintTop_toTopOf="@+id/constraintLayout"
        app:layout_constraintBottom_toBottomOf="@+id/constraintLayout"/>

    <Button
        android:text="Bias"
        app:layout_constraintEnd_toEndOf="@id/constraintLayout"
        app:layout_constraintStart_toStartOf="@id/constraintLayout"
        app:layout_constraintTop_toTopOf="@+id/constraintLayout"
        app:layout_constraintBottom_toBottomOf="@+id/constraintLayout"
        app:layout_constraintHorizontal_bias="0.25"
        app:layout_constraintVertical_bias="0.25"/>
</android.support.constraint.ConstraintLayout>
```

最终的效果是在页面中，“中心按钮”位于页面的中部，“比例按钮”位于页面的左上测，如图 2-13 所示。

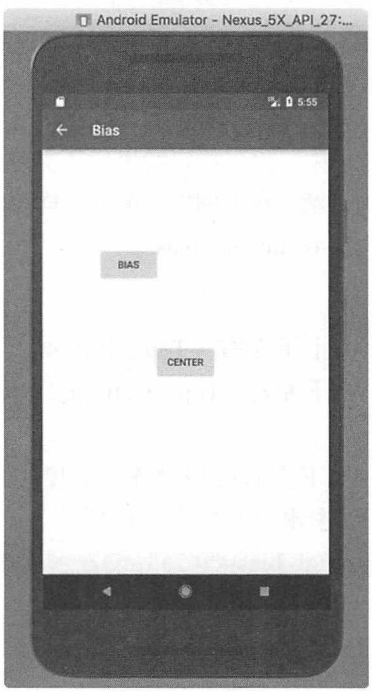


图 2-13　设置比例效果

3. 引导线约束

在 ConstraintLayout 约束布局中，除了支持与视图之间约束以外，还支持与引导线（Guideline）之间的约束。在本例中，单击首页 MainActivity 的 GuideLine 项，即可跳转展示与引导线的约束页面，即布局 R.layout.layout_guide_line。

在布局 layout_base 中：

- ◎ 引导线（guideLine）。
- ◎ orientation="vertical"：表示引导线的方向竖直。
- ◎ layout_constraintGuide_begin="72dp"：表示距离页面的左侧 72dp。
- ◎ 两个按钮，即“Guide Up”和“Guide Down”，左侧以引导线的左侧为约束，上下以父布局的上下为约束，使用比例方式排列，其中一个的比例是 0.25，另一个是 0.75。

当仅指定一个约束时，如仅指定左侧约束，未指定右侧约束，则表示控件紧贴在约束控件的一侧，如左侧。当同时指定两个约束时，则表示控件位于两侧约束控件的中心位置。引导线是虚拟的线不会显示，仅仅作为排列其他实体控件的约束。

```
<android.support.constraint.ConstraintLayout
    android:id="@+id/constraintLayout">
    <android.support.constraint.Guideline
        android:id="@+id/guideLine"
        android:orientation="vertical"
        app:layout_constraintGuide_begin="72dp"/>

    <Button
        android:text="Guide Up"
        app:layout_constraintStart_toStartOf="@id/guideLine"
        app:layout_constraintTop_toTopOf="@+id/constraintLayout"
        app:layout_constraintBottom_toBottomOf="@+id/constraintLayout"
        app:layout_constraintVertical_bias="0.25"/>

    <Button
        android:text="Guide Down"
        app:layout_constraintStart_toStartOf="@id/guideLine"
        app:layout_constraintTop_toTopOf="@+id/constraintLayout"
        app:layout_constraintBottom_toBottomOf="@+id/constraintLayout"
        app:layout_constraintVertical_bias="0.75"/>
</android.support.constraint.ConstraintLayout>
```

最终的效果是在页面中，“Guide Up”和“Guide Down”按钮位于距页面左侧 75dp 的位置，其中一个按钮在上，另一个在下，如图 2-14 所示。

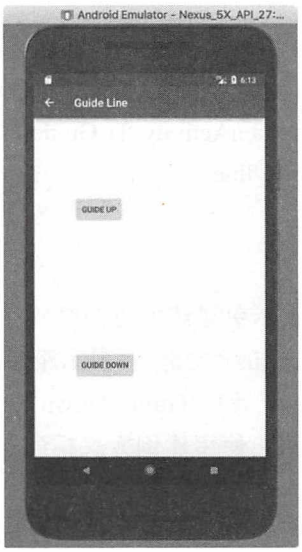


图 2-14　引导线约束效果

4. 控件填充

在 ConstraintLayout 约束布局中，支持自动填充宽高，把宽高设置为 0dp 时，会根据位置自动填充。在本例中，点击首页 MainActivity 的 Measure 项，即可跳转展示自动填充宽高页面，即布局 R.layout.layout_measure。

在布局 layout_measure 中：

◎ “小按钮”（Small）的左侧与父容器的左侧对齐，顶部与父容器的顶部对齐，即在页面的左上角；按钮的大小根据内容自动设置为最小，即宽高的属性值是 wrap_content。

◎ “大按钮”（Large）的底部和顶部与父容器的底部和顶部对齐，即在父容器的竖直中心位置；左侧与“小按钮”的右侧对齐；高度属性值 layout_height 设置为最小值 wrap_content；宽度属性值 layout_width 设置为 0dp，则表示宽度为从控件约束的左侧，一直到控件约束的右侧，填充全部空位，而不是真正的 0dp。

注意：只有当父布局是约束布局 ConstraintLayout 时，同时指定两侧约束，则宽高的 0dp 会填充至全部空白；当指定一侧约束或未指定约束时，则宽高的 0dp 表示 wrap_content 属性。也就是说，在约束布局 ConstraintLayout 中，不存在 0dp 的情况，如果需要不显示控件，则直接删除或将显示属性（visibility）设置为隐藏（gone 或 invisible）即可。

```
<android.support.constraint.ConstraintLayout
    android:id="@+id/constraintLayout">
    <Button
        android:id="@+id/small"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Small"
        app:layout_constraintStart_toStartOf="@id/constraintLayout"
        app:layout_constraintTop_toTopOf="@id/constraintLayout"/>

    <Button
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:text="Large"
        app:layout_constraintBottom_toBottomOf="@id/constraintLayout"
        app:layout_constraintEnd_toEndOf="@id/constraintLayout"
        app:layout_constraintStart_toEndOf="@id/small"
        app:layout_constraintTop_toTopOf="@id/constraintLayout"/>
</android.support.constraint.ConstraintLayout>
```

最终的效果是在页面中，小按钮位于页面的左上角；大按钮位于页面的竖直中心，按钮长度是从小按钮的右侧位置一直填充至页面的右侧，如图 2-15 所示。

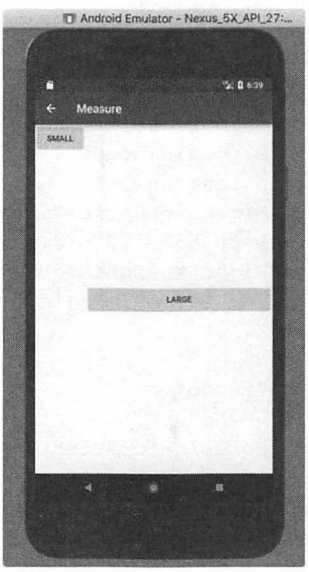


图 2-15　控件填充效果

5. 控件宽高比

在 ConstraintLayout 约束布局中，支持设置布局中控件的宽高比。在本例中，点击首页 MainActivity 的 Aspect Ratio 项，即可跳转展示控件的宽高比页面，即布局 R.layout.layout_aspect_ratio。

在布局 layout_aspect_ratio 中：

◎ 顶部图片：左侧和右侧与父容器的左侧和右侧对齐，即位于布局的水平中心；顶部和父容器的顶部对齐，即位于布局的顶部；高度 layout_height 设置为 200dp；宽度 layout_width 设置为 0dp，同时通过 layout_constraintDimensionRatio 设置宽高比为“16：9”，宽度值随着高度值的变化而变化。

◎ 底部图片：位置与顶部图片类似，只不过底部和父容器的底部对齐，即位于布局的底部；宽度 layout_width 设置为 200dp；高度设置为 0dp，同时宽高比设置为“4：3”。

ConstraintLayout 约束布局中控件的宽高比特性，使开发变得简洁，因为一般而言，图片尺寸都会选择合适的比例用于展示，如 16：9 或 4：3。通过宽高比，只需设置一个维度，另一个维度会自动计算，避免开发过程中不断地修改和计算。

```
<android.support.constraint.ConstraintLayout
    android:id="@+id/constraintLayout">
    <ImageView
        android:layout_width="0dp"
        android:layout_height="200dp"
        android:background="@color/colorAccent"
        android:src="@drawable/total_large"
        android:contentDescription="@null"
        app:layout_constraintDimensionRatio="16:9"
        app:layout_constraintLeft_toLeftOf="@+id/constraintLayout"
        app:layout_constraintRight_toRightOf="@+id/constraintLayout"
        app:layout_constraintTop_toTopOf="@+id/constraintLayout"/>

    <ImageView
        android:layout_width="200dp"
        android:layout_height="0dp"
        android:background="@color/colorAccent"
        android:contentDescription="@null"
        android:src="@drawable/total_large"
        app:layout_constraintBottom_toBottomOf="@+id/constraintLayout"
        app:layout_constraintDimensionRatio="4:3"
        app:layout_constraintLeft_toLeftOf="@+id/constraintLayout"
        app:layout_constraintRight_toRightOf="@+id/constraintLayout"/>
</android.support.constraint.ConstraintLayout>
```


最终的效果是在页面中，顶部图片的比例是 16：9，底部图片的比例是 4：3，图片的背景均用水粉色填充，如图 2-16 所示。

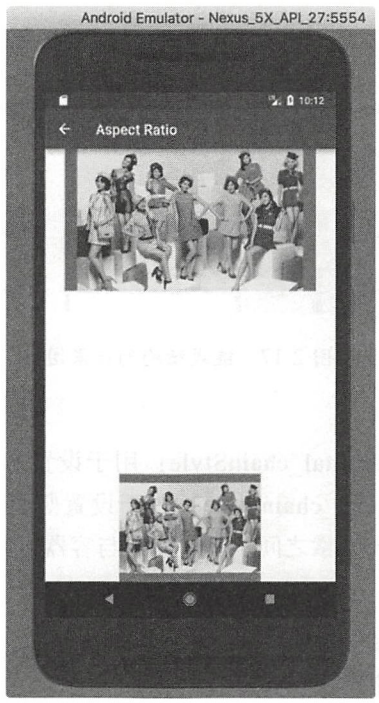


图 2-16　控件宽高比

2.3.3　链式结构

在 ConstraintLayout 约束布局中，还支持多个视图的排列组合，即链式结构。这与 LinearLayout 中多个控件排列的 layout_weight 属性非常类似，通过设置不同的链式结构，就可以排列出不同的样式。链式结构非常实用，也非常有趣。

◎ Spread Chain：元素之间留有空隙，链的两侧也留有空隙，空隙的宽度相等，即两侧含有空隙的等宽排列。

◎ Spread Inside Chain：元素之间留有空隙，链的两侧紧贴于容器两侧，空隙的宽度相等，即两侧不含空隙的等宽排列。

◎ Weighted Chain：元素之间不含空隙，紧密排列，宽度比例各不相同。

◎ Packed Chain：元素之间不含空隙，而链的两侧留有空隙，且空隙宽度相等。

◎ Packed Chain with Bias：元素之间不含空隙，而链的两侧留有空隙，但空隙宽度

不同。

链式排列不仅限于水平结构，也适用于竖直结构。链式结构的效果图如图 2-17 所示。

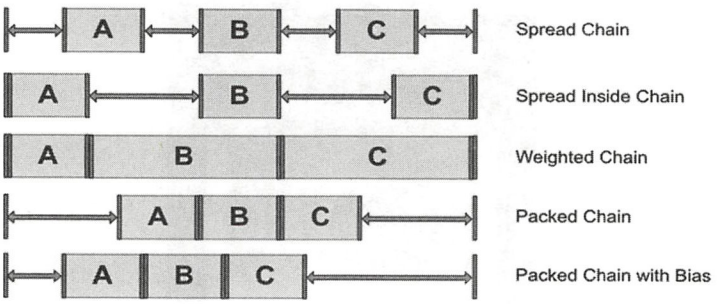


图 2-17　链式结构的效果图

例如：

◎　layout_constraintHorizontal_chainStyle：用于设置水平链。

◎　layout_constraintVertical_chainStyle：用于设置竖直链。

对于链式排列，需要元素与元素之间、首尾元素与容器之间的相互约束。假如水平链中含有 ABC 三个元素，则：

◎　A 的左侧与父容器的左侧约束。

◎　A 的右侧与 B 的左侧约束。

◎　B 的左侧与 A 的右侧约束。

◎　B 的右侧与 C 的左侧约束。

◎　C 的左侧与 B 的右侧约束。

◎　C 的右侧与父容器的右侧约束。

即形成一个完整的链式关系，同时，在元素中声明所需的链式样式，则自动形成所需的链式结构。

通过不同的链式组合，就可以生成更为复杂的视图样式，如图 2-18 所示。

ConstraintLayout 约束布局是 Android 系统中最优秀的布局设计之一，兼顾 LinearLayout 与 RelativeLayout 的优点，非常适合构建复杂布局，在布局设计中具有里程碑意义。在以往的开发过程中，RelativeLayout 虽然可以降低层级关系，但是内部控件之间非常耦合，一旦开发完成就很难修改，牵一发而动全身。同时，RelativeLayout 也不具备相对位置的功能，只能通过绝对值的形式进行开发，这样经常无法适配不同的分辨率。如果要引入相对位置，则必须增加 LinearLayout，由其提供 layout_weight 属性，进行等比例切分等功能，这样布局的层级数又会急剧增加，增加渲染时间。所以，对于复杂的布局而言，RelativeLayout 和 LinearLayout 几乎无

法完美地解决层级过多和相对位置之间的冲突问题。

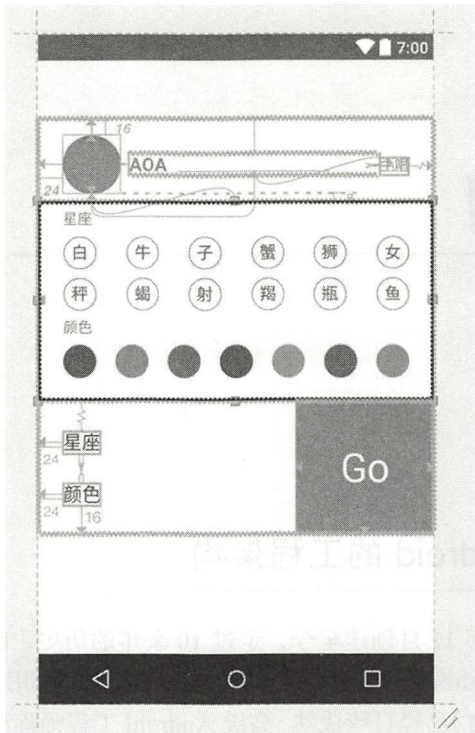


图 2-18　复杂的视图样式

因此，ConstraintLayout 约束布局的作用非常突出，兼顾两者的优点，是开发者急需的布局设计工具，通过约束布局，既可以设计出优美的布局，又可以降低布局的层级，加快渲染速度。高级的 Android 工程师需要在开发中找到问题的痛点，并通过合适的工具解决痛点，既考虑需求的完整性，又兼顾产品的内在性能。

第 3 章
项目架构

3.1 顶层设计 Android 的工程架构

Android 系统于 2007 年 11 月创建至今，走过 10 余年的历史，随着搭载 Android 系统的硬件平台逐渐增多，以及 Android 开发者的不懈努力，其系统的应用程序编程接口（Application Programming Interface，API）已经日臻成熟。高级 Android 工程师逐渐从如何更快地开发功能，转变为如何更稳定地开发功能，这涉及软件工程中的一些知识。通过某种工程架构，将不同的代码按照特定的结构组织起来，将关联的逻辑聚合在一起，当改进和定制功能时，不需要重新编写逻辑，尽量重用底层代码，以减少对原有系统造成影响。对于一个优秀的工程架构，需要满足以下软件工程的 8 个标准：

1）可靠性，通过拆分和组合逻辑，将相同功能聚合成模块，避免相互影响。

2）安全性，封装对外暴露的接口，提供统一和标准的访问方式，增强项目的安全防护。

3）伸缩性，在使用频率和使用人数急剧增加时，维持合理的性能和错误率。

4）定制化，根据不同的市场和变化的需求，增加或减少某些功能，适应环境的变化。

5）扩展性，当底层系统更新和新的接口增加时，支持升级系统，允许导入接口，并且兼容现有功能。

6）维护性，当系统出现错误时，能够得到及时定位与快速反馈，并且易于修复错误。

7）用户体验，给予用户良好的体验，既包括优美的界面，又包括良好的性能。

8）版本迭代，支持不断地调整已有的功能、添加新的功能，在不破坏架构的情况下，满足用户需求。

从另一个角度来看，对于一个优秀的工程架构，还需要满足以下面向对象的 5 个原则，就是经典的“SOLID”，将每个原则浓缩成一句话表示，即：

1）单一职责原则（Single Responsibility），每个方法或类有且仅有一个改变的理由。

2）开放/封闭原则（Open/Close），对于程序、方法和类，开放扩展或继承，关闭修改。

3）里氏替换原则（Liskov Substitution），当用超类的对象代替子类的对象时，不会破坏程序逻辑。

4）接口分离原则（Interface Segregation），不强制代码依赖于不使用的接口。

5）依赖倒置原则（Dependency Inversion），代码取决于抽象概念，而不是具体实现；抽象不依赖于细节，而细节依赖于抽象。

这些软件工程的标准和面向对象的准则，同样适用于 Android 的工程架构。随着 Android 的项目越来越成熟，项目的工程也越来越复杂，面对庞大的代码量，使用工程架构是必然的趋势。在开发实践中，工程师不断摸索，从已有 MVC（Model View Controller）架构中受到启发，陆续推演出 MVP（Model View Presenter），以及 MVVM（Model View ViewModel）等更适用于 Android 的工程架构。这些工程架构，对于困扰大型 Android 项目的若干棘手问题，提供了一定的解决方案，如：

1）因未分离视图逻辑与业务逻辑，所导致的逻辑爆炸。

2）版本迭代导致需要修改较多的类，代码隐藏潜在风险。

3）模块之间高耦合低内聚，相互关联，无法编码测试单元，测试独立功能。

目前，最为成功的工程架构都来源于早期的 MVC 模型，本节将由近及远地分析 MVC、MVP、MVVM 三个主流的工程架构，其他的工程架构基本来源于这三类。同时，根据这些架构不同的组织形式，讲解这样设计的优缺点，并进行横向对比，展示出一条完整的 Android 工程架构演进链。读完本节内容，可以高屋建瓴地设计出一个针对大型 Android 项目的工程架构。

同时，也可以参考 Google 的 Android 项目组开源的 android-architecture 工程，网址为 https://github.com/googlesamples/android-architecture。

3.1.1　MVC 架构

MVC 架构，中文译为模型、视图、控制器，其主要的设计思想基于视图逻辑的修改会多于业务逻辑的修改，当仅仅修改业务逻辑时，不会改变视图逻辑的代码，减少了修改代码导致的

潜在风险。MVC 架构（见图 3-1）将工程分为三个部分：数据层（又称模型层），即 Model；视图层（又称 UI 层），即 View；逻辑层（又称控制层），即 Contoller。

1）数据层：负责管理视图逻辑与业务逻辑，提供网络与数据库的监听和修改接口。

2）视图层：负责显示来自于数据层的数据，接收用户的交互信息。

3）逻辑层：根据来自视图层的交互信息，或定时任务，指挥数据层更新数据，或指挥视图层重载数据层的数据。

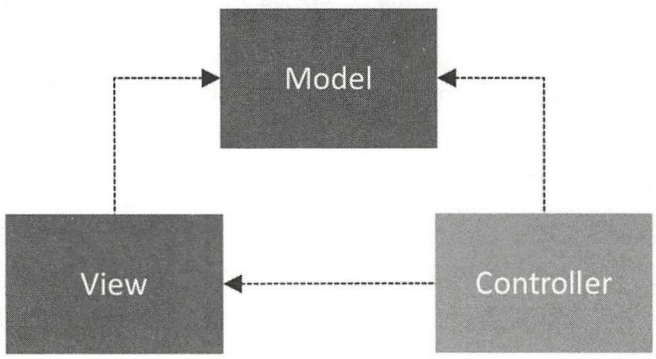


图 3-1　MVC 架构

根据 MVC 架构，视图层依赖于数据层，逻辑层依赖于视图层和数据层，数据层完全独立。视图层作为整个架构的窗口，显示数据和接受用户响应事件，根据控制层的指令，被动地显示数据层的数据。逻辑层作为整个架构的大脑，根据来源于视图层的用户响应事件，或定时任务，给数据层发出指令更新数据以及给视图层发出指令重新加载数据。数据层作为整个架构的核心，提供数据与更新数据，视图层和逻辑层都会依赖于数据层，视图层的显示来源于数据层的数据，逻辑层也会更新数据层的数据。最终，数据层完全从项目中解耦，数据层中的视图逻辑与业务逻辑也从项目中独立，不依赖于显示。数据层支持编写独立的测试单元，测试视图逻辑与业务逻辑的正确性。

逻辑层负责更新数据层的数据，根据更新的方式不同，即主动或被动，MVC 架构又分为被动（Passive）模式和主动（Active）模式。

1．MVC 架构的被动模式

在 MVC 架构的被动模式中，逻辑层是唯一操作数据层的类，可以修改数据层的数据，而视图层只能根据逻辑层的指令，被动地从数据层中获取数据。基于视图层的用户响应事件（handleEvent），逻辑层指挥数据层更新数据（updateModel）。在数据更新完成后，逻辑层指挥视图层更新界面（update），视图层从数据层获取更新后的数据（getData），重新加载，完成一

个完整的响应事件逻辑，如图 3-2 所示。在被动模式中，逻辑层控制整个流程的更新过程，根据用户响应事件，被动地指挥数据层与视图层，完成更新数据与加载数据的功能。

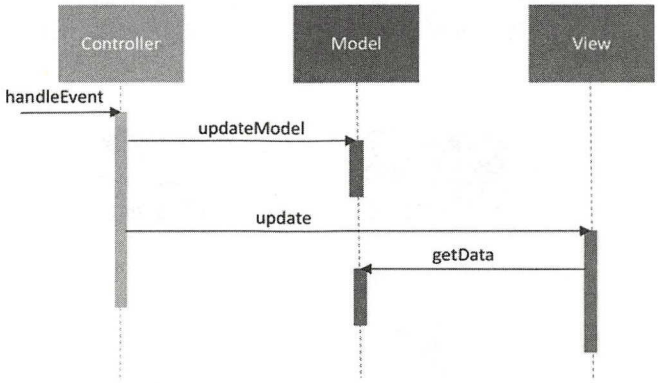


图 3-2　MVC 架构的被动模式

2．MVC 架构的主动模式

在 MVC 架构的主动模式中，逻辑层不再是唯一指挥视图层的类。在数据层中，存在自动更新当前数据状态的观察者（Observer）模块，由此模块通知视图层，获取数据层中更新后的数据，重新加载，如图 3-3 所示。观察者模块依赖于数据层，视图层与逻辑层依赖于观察者模块。观察者模块根据数据层数据的更新状态，发送相应的广播，通知所有关心数据层数据的观察者。视图层是注册在观察者模块的一个观察者，接收模块发送的广播。逻辑层指挥数据层的更新操作，间接指挥观察者模块。

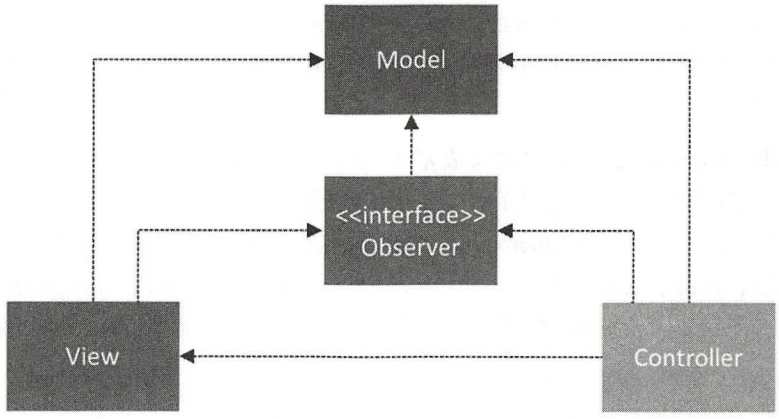


图 3-3　MVC 架构的主动模式

主动模式与被动模式的不同在于，当数据层完成数据更新时，不再通过逻辑层指挥视图层更新数据，而是在数据层内部调用观察者模式（notify），通知视图层重新加载数据层的数据，如图 3-4 所示。视图层实现观察接口，在数据层的观察者模块中，注册成为一个观察者，当数据层的数据被修改时，数据层通过观察者模块发送广播，通知视图层的所有观察者，执行相应的操作。

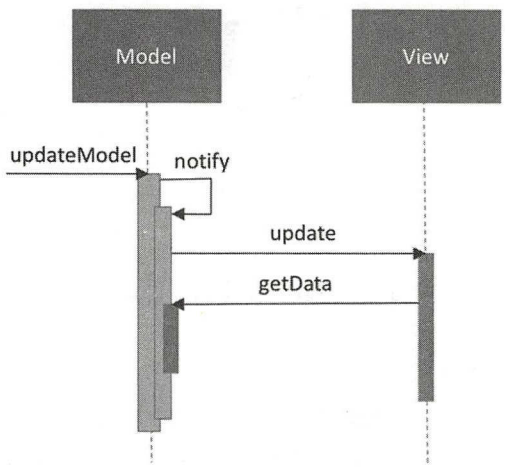


图 3-4　Active MVC

3．MVC 架构的应用范围

在未使用任何架构的 Android 项目中，Activity（或 Fragment）的功能，既是视图层、逻辑层，又是数据层，并未进行有效的分离，导致核心页面逻辑冗余，混杂着显示、视图逻辑与业务逻辑，代码行数众多。当需要修改或添加某些功能时，极其容易引入无法测试的 Bug。一些含有初级架构思想的团队，从项目中剥离出数据层，即把视图逻辑与业务逻辑独立，编写针对于业务层的测试单元。

那么如何将 Android 项目重构改写成基于 MVC 架构呢？首先，从项目的 Activity（或 Fragment）中抽取视图逻辑与业务逻辑，管理网络数据与本地数据，这样就分离出了数据层；其次，从项目的 Activity（或 Fragment）中，再抽取控制逻辑，指挥页面与数据的更新，这样就分离出了逻辑层；最终，在项目的 Activity（或 Fragment）中，只剩下与界面显示相关的逻辑，这样就形成了视图层。至此，MVC 的三层，数据层、逻辑层、视图层，已经全部呈现。

具体实现 MVC 架构的难点在于，如何处理当更新数据时三个模块之间的联动。在 MVC 架构的被动模式中，视图层的类继承于视图层基类（如 BaseView），逻辑层的类绑定视图层基类的引用，通过接口，指挥视图层更新数据；在 MVC 架构的主动模式中，数据层使用观察者模

式，视图层注册加入观察者队列，广播通知更新数据。

> 注意：在逻辑层与数据层中的代码，不含有任何显示部分，即不会引用 Android 的 Context 类，支持编写不含 Android 界面的测试单元。

对比未使用 MVC 结构的 Android 项目，基于 MVC 架构的项目更容易保持代码的一致性，减少文件的修改次数，当修改视图逻辑时，较少地修改数据层的代码；当修改业务逻辑时，较少地修改视图层的代码。

MVC 架构分离显示部分（视图层）与逻辑部分（数据层），增强项目的可测试性与可扩展性。数据层从项目中独立，不引入任何 Android 的界面类，支持单元测试(Unit Test)；逻辑层仅仅含有视图层的引用，也不引入任何 Android 的界面类，同样支持单元测试。视图层完全满足单一职责原则（SRP），将用户响应事件传递给逻辑层，受逻辑层的指挥，展示数据层的数据，其中不包含任何逻辑，支持界面（UI）测试。经过 MVC 架构重构过的 Android 项目，在可测试性方面，有了极大的提高。

4．MVC 架构的缺陷

所有架构都不适用于迷你项目，在基于架构分离职责时，必然会引入若干的控制单元，导致类的增多，即设计过度。但是在大型项目中，良好的项目架构可以避免风险，减少因新增或修改导致的不可控因素，提高项目的稳定性。这个理论同样适用于设计模式。

MVC 架构也存在一些不可避免的问题。视图逻辑与业务逻辑并未分离，都集中在数据层，因为视图层只负责显示，数据层为视图层提供数据，数据层既需要管理数据，又需要承担视图逻辑与业务逻辑，降低了整体架构的灵活性。对于视图逻辑与业务逻辑的区分，当需要计算某个时间时，属于业务逻辑，如毫秒值；当需要显示这个时间时，属于视图逻辑，既可以是“XXXX 年 XX 月 XX 日”，也可以是“XXXX-XX-XX”。在视图逻辑中，视图层负责显示，数据层负责格式，职责部分重叠，两个部分过于耦合。

无论是在 MVC 架构的被动模式，还是主动模式中，视图逻辑都会产生若干问题。在被动模式中，逻辑层通知数据层更新数据，通知视图层显示数据。视图逻辑是耦合的关键症结，默认由数据层处理，则隐式地依赖于视图层，当视图层修改显示样式时，数据层也需要修改视图逻辑，导致两个模块的耦合度增加；如果改为视图层处理，则单元测试会遗漏这些视图逻辑。

如显示医生的名字与科室，如果从数据层分别取出，在视图层拼接（视图逻辑），则属于视图逻辑，位于视图层；如果从数据层直接取出拼接（视图逻辑）后的数据，则属于视图逻辑，位于数据层。前者会导致当测试数据层时遗漏视图逻辑，后者会导致当视图层修改显示样式时，数据层也需要修改对应的视图逻辑。为便于读者理解，只给出一个小例子，真实情况将复杂得多。

```
// 视图层处理视图逻辑
String docName = userModel.getName();
String docClinic = userModel.getClinic();
nameTextView.setText(docName + ", " + docClinic)

// 数据层处理视图逻辑
String nameAndClinic = userModel.getNameAndClinic();
nameTextView.setText(nameAndClinic);
```

在主动模式中，除了耦合度的问题，还有观察者模块的数量膨胀问题。在数据层中，每个小的视图逻辑都需要增加观察者模块，来保证视图层更新顺利地进行，导致需要大量观察者模块。

因为 MVC 架构未解决业务逻辑与视图逻辑的分离问题，而且对于视图逻辑的处理，一般是增加耦合或者遗漏测试，所以架构师们又设计出 MVC 架构的进化版 MVP（Model View Presenter）架构。

3.1.2 MVP 架构

MVP 架构，中文译为模型、视图、展示，将工程分为 3 个部分：数据层（又称模型层），即 Model；视图层（又称 UI 层），即 View；展示层（又称表示层），即 Presenter。其主要的设计思想是：当数据层传输数据至视图层时，使用展示层在数据上封装视图逻辑，再传递给视图层，如图 3-5 所示。相对于 MVC 架构，MVP 架构将视图逻辑从数据层移至展示层，数据层仅仅保留业务逻辑，实现视图逻辑与业务逻辑的分离。

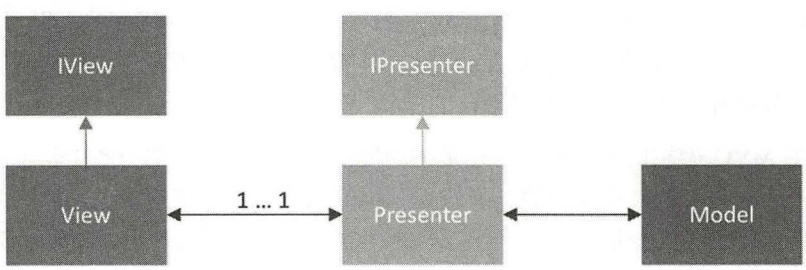


图 3-5 MVP 架构

1）数据层（Model）：负责管理业务逻辑，提供网络与数据库的监听和修改接口。

2）视图层（View）：负责显示来自于展示层的数据，接收用户的交互信息。

3）展示层（Presenter）：负责管理视图逻辑与交互逻辑，根据来自视图层的交互信息或定

时任务，指挥数据层更新数据；接收数据层的数据，封装视图逻辑，为视图层提供显示数据。

视图层与展示层的关系非常紧密，相互引用，并且一一对应，即每个视图层类都会对应一个展示层。视图层将界面显示与用户事件响应封装成接口，供展示层调用；展示层将视图逻辑也封装成接口，供视图层调用。视图层处理 Android 界面逻辑，展示层处理视图逻辑，视图层使用 UI 测试，展示层使用单元测试。

为了便于管理和查找，创建视图层和展示层的父接口，用于相互关联，具体接口都继承于父接口，在接口中放入具体方法，再将配对的视图层和展示层的具体接口放置在一个合同类中，表明这一组视图接口与展示接口是一一对应的。

```
public interface TasksContract {
    // View 的接口类
    interface View extends BaseView<Presenter> {
    }
    // Presenter 的接口类
    interface Presenter extends BasePresenter {
    }
}

// View 的基类
public interface BaseView<T> {
    void setPresenter(T presenter);
}

// Presenter 的基类
public interface BasePresenter {
    void start();
}
```

1. MVP 架构的数据层

数据层负责获取或储存在远程或本地的数据，操作方式有获取和储存两种；数据来源有远程和本地两种。一个常见的例子是，当处理数据的显示逻辑时，数据层会优先检索本地数据，如果存在，则直接返回数据；如果不存在，则请求网络数据，完成后同步至本地，再返回数据。在数据层的构造器中，绑定本地数据源（tasksLocalDataSource）与远程数据源（tasksRemoteDataSource）两个接口类，无论是本地还是远程，都继承于数据源的父接口类（TasksDataSource）。数据源的父接口类负责数据源的逻辑，规定若干操作数据源的通用接口；具体的本地或远程接口类负责通用接口的实现。

```
public static TasksRepository getInstance(
        TasksDataSource tasksRemoteDataSource,
        TasksDataSource tasksLocalDataSource
) {
    if (INSTANCE == null) {
```

```
            INSTANCE = new TasksRepository(tasksRemoteDataSource, tasksLocalDataSource);
        }
        return INSTANCE;
    }
```

2．MVP 架构的视图层

视图层与展示层配合使用，显示展示层提供的数据，将用户响应事件反馈至展示层。当分离视图逻辑后，Activity（或 Fragment）就是视图层。视图层需要绑定特定的展示层，视图层接口（View）继承于含有展示层接口参数（Presenter）的视图层父接口（BaseView），并提供绑定展示层方法的接口（setPresenter()）。

```
    public interface TasksContract {
        interface View extends BaseView<Presenter> {
        }
    }
    public interface BaseView<T> {
        void setPresenter(T presenter); // 绑定展示层
    }
```

在经典的 Activity-Fragment 模式中，视图层就是 Fragment（TaskDetailFragment），继承视图层接口（TaskDetailContract.View），实现接口中的方法。

```
    public class TaskDetailFragment extends Fragment implements TaskDetailContract.
View {
    }
```

同时，在 Fragment 中，重写 setPresenter()方法绑定展示层，同时，在 onResume()中执行展示层的数据初始化方法，即 mPresenter.start()。

```
    @Override
    public void setPresenter(@NonNull TaskDetailContract.Presenter presenter) {
        mPresenter = checkNotNull(presenter);
    }

    @Override
    public void onResume() {
        super.onResume();
        mPresenter.start();
    }
```

当响应用户事件时，视图层截获响应事件，通过展示层接口传递事件至展示层，由展示层负责处理；完成后，展示层通过视图层接口传递数据给视图层显示，由视图层负责显示。视图层与展示层通过接口紧密关联，两个层的接口一般也是相互对应的。

3．MVP 架构的展示层

展示层依附于视图层存在且一一对应，内部绑定视图层与数据层的引用。展示层类（Presenter）继承于展示层父类（BasePresenter），并提供数据初始化方法的接口（start()）。

```
public interface TasksContract {
    interface Presenter extends BasePresenter {
    }
}
public interface BasePresenter {
    void start(); // 启动数据加载
}
```

在展示层的构造器中，绑定作为参数传递的视图层，同时调用视图层绑定展示层的接口，完成相互绑定。除了视图层，在构造器中，还会绑定作为参数传递的数据层。所以，在展示层的构造器中至少含有两个参数，即视图层的引用和数据层的引用。

```
public TaskDetailPresenter(
        @Nullable String taskId,
        @NonNull TasksRepository tasksRepository,
        @NonNull TaskDetailContract.View taskDetailView
) {
    mTaskId = taskId;
    mTasksRepository = checkNotNull(tasksRepository, "tasksRepository cannot be
null!"); // 绑定数据层
    mTaskDetailView = checkNotNull(taskDetailView, "taskDetailView cannot be
null!"); // 绑定视图层

    mTaskDetailView.setPresenter(this); // 调用视图层绑定当前展示层
}
```

当响应用户事件时，展示层请求数据层，获取数据，根据视图逻辑将数据封装，通过接口，在视图层中显示数据。展示层的接口覆盖全部的事件处理逻辑，每一个视图层的事件都可以在展示层中获得响应，并且封装视图逻辑。

4．MVP 架构的优劣

MVP 架构是针对 Android 项目的设计模式，对于已分离数据层的初级架构项目容易重构，根据每个 Activity（或 Fragment）创建对应的展示层，将视图逻辑从视图层（如 Activity）和数据层中剥离出来，放置于展示层即可。这样，当修改业务逻辑时，仅仅修改数据层即可，视图层与展示层不动；当修改视图逻辑时，仅仅修改展示层即可，数据层与视图层不动；当修改页面显示样式时，仅仅修改视图层即可，数据层与展示层不动。MVP 架构的数据层、视图层、展示层，基于业务逻辑、显示样式、视图逻辑，完全解耦。数据层与展示层使用单元测试，视图层使用 UI 测试。对于迷你项目，MVP 架构仍会导致设计过度。除此之外，MVP 架构有如下三

个致命缺陷：

1）无论修改视图层，或修改数据层，都要修改展示层，违反了单一职责原则；

2）当页面过于复杂时，展示层包含大量视图逻辑，非常沉重；

3）对于相同的视图逻辑，不同的展示层无法共享，都需要包括。

MVP 架构的核心在于展示层，打破视图逻辑与业务逻辑之间的耦合，创建数据展示的通道，隔离视图逻辑，支持单元测试。但仍不完美，于是架构师们又设计出 MVP 架构的进化版 MVVM（Model View ViewModel）架构，来解决 MVP 架构的问题。

3.1.3 MVVM 架构

MVVM 架构，中文译为模型、视图、视图模型，将工程分为 3 个部分：数据层（又称模型层），即 Model；视图层（又称 UI 层），即 View；视图数据层，即 ViewModel。其主要的设计思想与 MVP 架构类似，都是分离视图逻辑与业务逻辑，视图层与数据层相同，不同的是视图数据层（ViewModel），MVP 的展示层抽象出事件处理的接口，MVVM 的视图数据层为事件处理提供数据流，如图 3-6 所示。相比之下，视图数据层比展示层的粒度更细。

1）数据层：负责管理业务逻辑，提供网络与数据库的监听和修改接口，与视图数据层（ViewModel）区分，又称为 DataModel，纯数据层。

2）视图层：负责显示来自于多个视图数据层的数据，接收用户的交互信息。

3）视图数据层：负责管理视图逻辑交互逻辑，将逻辑分类，管理一类特定的逻辑，与多个视图层绑定，为其提供特定的数据支持与交互处理。

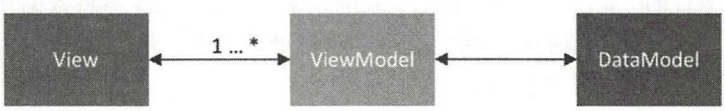


图 3-6　MVVM 架构

MVVM 架构与 MVP 架构的结构相似，核心目标都是分离显示界面与视图逻辑，但是在实现方式上，两者有如下差异：

1）在 MVP 架构中，展示层与视图层是强绑定的关系，每个展示层都与被服务的视图层对应，为其提供数据，是一对一的关系；在 MVVM 架构中，视图数据层仅仅提供特定的数据流，相同的数据流可以被多个视图层使用，视图数据层与视图层是弱绑定的关系，即一对多的关系。

2）在 MVP 架构中，展示层与视图层相互引用，配合使用；在 MVVM 架构中，视图数据层独立于视图层，而视图层保留视图数据层的引用。

3）在 MVP 架构中，展示层与视图层是一个组合，共同配合显示页面；在 MVVM 架构中，视图数据层与视图层的关系类似于生产者（视图数据层）和消费者（视图层）的关系。消费者知道生产者，而生产者只负责提供数据，并不关心谁来消费这些数据。

1．MVVM 架构的数据层

MVVM 广义上包含两个数据层：数据层，又称为纯数据层（DataModel），与视图数据层区分。数据层通过事件流，提供多种数据源，如网络数据、数据库数据、首选项（Shared Preferences）数据等，并负责全部的业务逻辑。数据层提供泛化数据的接口，确保业务逻辑的独立完整，可以被多个页面共享与使用，并为视图数据层提供数据。数据层不含有 Android 的页面代码，支持单元测试。

其工作原理与 MVP 模式的数据层类似。

2．MVVM 架构的视图数据层

对于视图数据层而言，与视图层的关联多于数据层，从数据层中获取必要的数据，在数据之上封装视图逻辑，为不同的视图层提供完整的展示数据。对于完整而言，主要包含两个方面：

1）视图数据层为视图层提供完整状态的数据，不需要任何加工。如 MVC 架构中的例子，数据层提供医生姓名与科室，当需要展示“医生姓名+医生科室”时，视图数据层会提供拼接后的数据，而不是单独数据。当视图逻辑需要修改时，通过广播将全部视图层的数据更新至最新。

2）视图层仅仅负责显示数据，不含有任何视图逻辑与时间处理，这些全部交由视图数据层处理。

视图数据层一般使用观察者模式，与 DataBinding 配合使用，继承观察者模式的基类（BaseObservable），将视图逻辑的属性用观察者域（ObservableField）表示，当属性发生变化时，更新全部显示属性的页面。

```
public abstract class ViewModel extends BaseObservable {
    public final ObservableField<String> testText = new ObservableField<>();
}
```

在布局（Layout）中，通过 DataBinding 绑定视图数据层。

```
<data>

    <import type="android.view.View" />

    <variable
        name="viewmodel"
        type="xxx.ViewModel" />
</data>
```

视图数据层在数据层之上封装视图逻辑，并不含有 Android 界面类，支持单元测试。数据视图层所绑定的数据层也支持自由替换，比如将线上数据源替换为模拟的测试数据源，验证视图逻辑的可用性。

```
public class ViewModel {
    private final IDataModel mDataModel;

    public ViewModel(IDataModel dataModel) {
        mDataModel = dataModel;
    }

    public Observable<Data> getData() {
        return mDataModel.getData();
    }
}
```

3. MVVM 架构的视图层

视图层负责显示视图数据层的数据，同时通知视图数据层处理用户响应事件，因此，视图层需要绑定数据视图层，调用数据视图层的接口。如经典的 Activity-Fragment 模型中，在 onCreate()方法中，将数据视图层绑定至 Fragment。

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // 创建 Fragment
    TaskDetailFragment taskDetailFragment = findOrCreateViewFragment();

    // 创建 ViewModel
    mTaskViewModel = findOrCreateViewModel();

    // 将 ViewModel 绑定至 Fragment
    taskDetailFragment.setViewModel(mTaskViewModel);
}
```

当 Fragment 接收到用户的响应事件时，调用数据视图层的事件处理接口，当事件执行完成后，通过观察者模式广播更新后的数据，视图层重新加载。

```
private void setupFab() {
    FloatingActionButton fab =
            (FloatingActionButton)
getActivity().findViewById(R.id.fab_edit_task);

    fab.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
```

```
            mViewModel.startEditTask();
        }
    });
}
```

MVVM 的重要特性是降低耦合度。Model 不含 Android 类，只含业务逻辑，支持单元测试；ViewModel 在 Model 上封装 UI 逻辑，不含 Android 类，支持单元测试。

4．MVVM 架构的优势

相对 MVC 架构与 MVP 架构，MVVM 架构的优势是进一步解耦视图逻辑与业务逻辑，使整个项目的耦合度降低，内聚度升高。MVVM 架构具有如下几个优势：

1）在 MVVM 架构中，视图层与视图数据层的耦合度，在 MVP 架构中，视图层与展示层的耦合度。

2）在 MVVM 架构中，视图层仅作为视图数据层的消费者，当修改显示页面时，不需要修改视图数据层。

3）在 MVVM 架构中，根据不同的业务组合，创建若干个高内聚视图模型层，支持视图层共享与替换视图模型层，实现逻辑的复用。

4）在 MVVM 架构中，彻底地分离显示页面与视图逻辑，使用 DataBinding 模式，将显示页面置于布局中，将视图逻辑置于视图数据层。

5）在 MVVM 架构中，视图层与视图数据层是一对多的关系，视图数据层独立于视图层；视图数据层与数据层是多对多的关系，数据层独立于视图数据层。

6）在 MVVM 架构中，视图数据层和数据层，与显示页面完全解耦，不含有任何 Android 的页面类，提高了可测试性。

MVC 架构作为早期的架构，来源于 Java 项目，在应用 Android 项目时，导致视图逻辑无法分离。MVC 架构的应用领域主要为大型 Java 项目，其较少修改视图逻辑，较多修改业务逻辑，主要保证业务逻辑的修改不影响视图逻辑，而对视图逻辑的修改关注较少。MVP 架构是 MVC 架构基于 Android 项目的改进，朴素地将已有的 Activity（或 Fragment）拆分成两个层，将视图逻辑与 MVC 架构的控制层都移入展示层，当修改视图逻辑时，不会影响到数据层，降低模块之间的耦合度，但是导致展示层过于臃肿，同时无法复用一些常用的视图逻辑。MVVM 架构在 MVP 架构的基础上，进一步解耦，把 MVP 架构中的视图层与展示层一对一的关系，拆分为视图层与视图数据层的一对多的关系，将含有视图逻辑的视图数据层完全独立出来，使整个项目的架构更加灵活。

目前工业界最常用的三类工程架构，即 MVC 架构、MVP 架构、MVVM 架构，均已细致

地阐述。读者通过理解原理，才能更好地将已有的项目重构，降低模块之间的耦合度，提升稳定性，避免因新增或修改功能带来的风险。

3.2 顶层设计基于 Flux 的流式架构

任何架构的最终目的都是让程序更加有序、便于扩展功能、容易追踪异常，Flux 架构也是如此。Flux 架构是 Facebook 用来构建用户端 Web 应用程序的架构体系，与其他形式的框架相比，更类似于一种架构思想，用于管理和控制应用中的数据流向。Flux 顾名思义就是“流”，Flux 架构是以数据流为基础的架构。这里的数据包括但不限于来自服务端的数据，如页面中 View 的一些状态，还有临时存储在本地，需要持久化到服务端的数据等。Flux 架构属于数据驱动型的架构，在以数据为核心的场景中非常适合。由 Facebook 开发的 ReactNative 移动端项目就是以 Flux 架构为核心，通过修改状态，更新页面视图。该项目已经开源，有兴趣的读者也可以阅读源码，了解其中的内容。

Flux 架构的基本框架如图 3-7 所示。

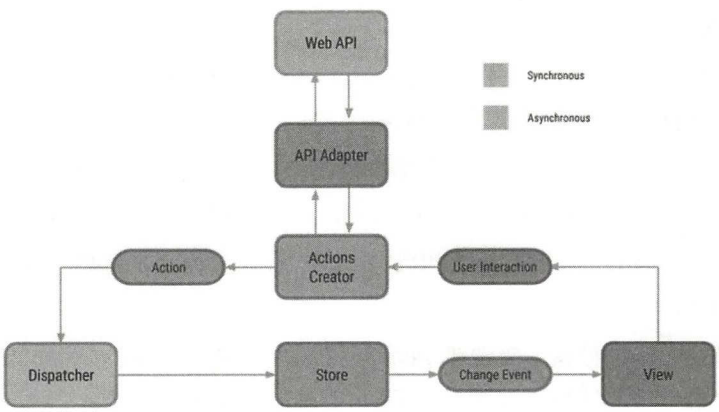


图 3-7 Flux 架构的基本框架

Flux 架构主要分为四个模块:

（1）视图：即 View，根据用户交互（User Interaction）的信息创建响应事件，调用行为创建器（ActionCreator），将信息转换为行为。

（2）行为创建器：即 ActionCreator，根据响应事件的类型和数据，创建行为（Action），把行为发送给调度器（Dispatcher）。同时，也可以执行网络操作（WebApi），当完成操作后再发送行为。

（3）调度器：即 Dispatcher，把接收到的行为通过总线（EventBus）系统发送出去，存储器（Store）根据行为的类型和数据修改数据模型。

（4）存储器：即 Store，维护特定的数据状态，接收总线分发的行为，根据行为的类型执行不同的业务逻辑。当完成数据操作时，发送修改事件（ChangeEvent），当视图接收到事件时，更新显示内容。

Flux 架构通过视图、行为创建器、调度器和存储器四个模块，实现数据流的闭环，响应用户事件。

掌握更多的设计架构，在实际应用中，可以根据业务场景的侧重点不同，选择合适的架构，做到事半功倍。同时，在程序设计中，也可以参考不同的架构思想，让所开发的模块具有更高的可扩展性。当然，Flux 架构思想也可以应用于移动端的开发，比如 Android 项目。本节会讲解如何基于 Flux 架构思想开发 Android 项目。本例是一个基于 Flux 架构的待办事项清单（ToDoList）应用，在页面中，用户触发各种增删改查（CRUD）的事件，通过 Flux 框架先修改数据状态，再反馈页面显示。

本文实例完整代码的下载地址为 https://github.com/SpikeKing/MyFluxApp-TodoList。

3.2.1　视图

在 Android 框架中，视图部分是一个 Activity，负责响应用户交互的事件，发生给行为创建器，同时接收修改事件，更新页面。

在 Activity 的启动接口 onCreate()中，设置页面的布局，将页面绑定至 ButterKnife，初始化 Flux 组件，设置页面的竖直列表视图 RecyclerView。Flux 的核心组件包括：调度器、行为创建器、存储器，三者都是单例形式，其中调度器依赖于事件总线类型，而行为创建器和存储器依赖于调度器。

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main); // 设置 Layout
    ButterKnife.bind(this); // 绑定 ButterKnife
    initDependencies(); // 创建 Flux 的核心管理类
    // 设置 RecyclerView
    mMainList.setLayoutManager(new LinearLayoutManager(this));
    mListAdapter = new RecyclerAdapter(sActionsCreator);
    mMainList.setAdapter(mListAdapter);
}

// 初始化：Dispatcher 调度器，Action 事件，Store 状态
```

```
private void initDependencies() {
    sDispatcher = Dispatcher.getInstance(new Bus());
    sActionsCreator = ActionsCreator.getInstance(sDispatcher);
    sTodoStore = TodoStore.getInstance(sDispatcher);
}
```

由于调度器的本质属于事件总线类型类型，在页面的显示和隐藏时，即 onResume()和 onPause()，需要注册和关闭注册。由于调度器需要向存储器发送事件，因而关联至存储器，即调度器和存储器相互依赖。

```
@Override
protected void onResume() {
    super.onResume();
    // 把订阅接口注册到 EventBus
    sDispatcher.register(this);
    sDispatcher.register(sTodoStore);
}

@Override
protected void onPause() {
    super.onPause();
    // 解除订阅接口
    sDispatcher.unregister(this);
    sDispatcher.unregister(sTodoStore);
}
```

视图的第一个功能是响应交互，接收用户交互的事件，本例的主要功能是操作 Todo 项，提供添加、选中和清除 Todo 项三种功能，即 addItem()、checkItem()、clearCompletedItems()，响应布局中的三个按钮事件。

- addItem()：添加项，即 addTodo()；重置输入框，即 resetMainInput()。
- checkItem()：改变全部项的选中状态，即 checkAll()。
- clearCompletedItems()：清除全选中的项，即 clearCompleted()；重置全选中按钮状态，即 resetMainCheck()。

其余的响应交互逻辑位于主页的列表中，即 RecyclerView 的 RecyclerAdapter 中，稍后分析。

```
// 添加 Todo 项按钮
@OnClick(R.id.main_add) void addItem() {
    addTodo(); // 添加 TodoItem
    resetMainInput(); // 重置输入框
}

// 选中 Todo 项按钮
@OnClick(R.id.main_checkbox) void checkItem() {
    checkAll(); // 所有 Item 项改变选中状态
}
```

```
// 清理完成的 Todo 事项
@OnClick(R.id.main_clear_completed) void clearCompletedItems() {
    clearCompleted(); // 清除选中的状态
    resetMainCheck();
}
```

主页面的三个交互按钮如图 3-8 所示。

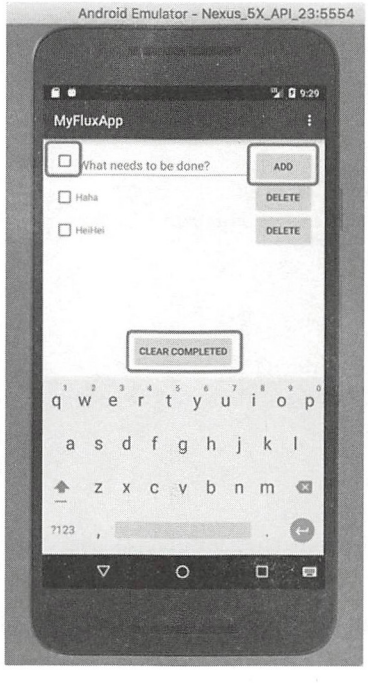


图 3-8　主页面的三个交互按钮

接着，具体分析响应交互逻辑，交互逻辑分为两类：第一类是不需要处理的纯页面改变，如重置输入框（resetMainInput）、全选中按钮的置换状态（resetMainCheck）等，第二类是需要处理的数据改变，如添加项（addTodo）、改变全部项的选中状态（clearCompleted）、清除全选中的项（clearCompleted）等。第一类事件页面自主处理，第二类事件全部交给行为创建器处理，视图仅仅作为代理，将必要的用户事件参数传递给行为创建器。

```
// 添加项，向 ActionsCreator 传递输入文本
private void addTodo() {
    if (validateInput()) {
        sActionsCreator.create(getInputText());
    }
}

// 重置输入框
```

```
    private void resetMainInput() {
        mMainInput.setText("");
    }

    // 改变全部项的选中状态(ActionsCreator)
    private void checkAll() {
        sActionsCreator.toggleCompleteAll();
    }
    // 清除全选中的项(ActionsCreator)
    private void clearCompleted() {
        sActionsCreator.destroyCompleted();
    }
    // 重置全选中按钮状态
    private void resetMainCheck() {
        if (mMainCheck.isChecked()) {
            mMainCheck.setChecked(false);
        }
    }

    // 验证输入框是否是空
    private boolean validateInput() {
        return !TextUtils.isEmpty(getInputText());
    }

    // 获取输入数据
    private String getInputText() {
        return mMainInput.getText().toString();
    }
}
```

视图的第二个功能是接收调度器的修改事件（ChangeEvent），更新页面。在 Activity 页面的 onResume()中，将当前页面注册于调度器，目的是接收调度器的广播。当数据存储状态改变时，即广播 TodoStoreChangeEvent 事件，当前页面更新 UI 的显示状态，即调用 updateUI()。页面不需关注改变具体的改变事件，无论什么事件，都从存储器（Store）中重新加载全部数据。当删除某项时，通过底部的 Snackbar 提示判断是否需要恢复，即撤销删除操作，当用户需要恢复时，点击文本，则调用行为创建器（ActionCreator）处理恢复事件，这也属于用户交互事件。

```
    // 接收事件的改变
    @Subscribe
    public void onTodoStoreChange(TodoStore.TodoStoreChangeEvent event) {
        updateUI();
    }

    // 更新 UI，核心方法
    private void updateUI() {
        // 设置适配器数据，每次更新 TodoStore 的状态
        mListAdapter.setItems(sTodoStore.getTodos());
```

```
        if (sTodoStore.canUndo()) { // 判断是否可恢复
            // 下面的提示条，恢复删除，提示信息
            Snackbar snackbar = Snackbar.make(mMainLayout, "Element deleted",
Snackbar.LENGTH_LONG);

            // 恢复按钮
            snackbar.setAction("Undo", new View.OnClickListener() {
                @Override
                public void onClick(View view) {
                    sActionsCreator.undoDestroy();
                }
            });
            snackbar.show();
        }
    }
```

Flux 架构的视图主要位于 Activity 页面中，接收用户交互事件，交给行为创建器处理。当处理完成后，由调度器通知，根据存储器的数据，更新页面状态。页面异步处理用户交互事件，将页面的业务逻辑与视图逻辑解耦。

本例的其余视图部分位于 RecyclerView 的 RecyclerAdapter 中。

3.2.2　行为创建器

行为创建器（ActionsCreator）提供接口，用于创建行为，根据行为匹配类型将数据字典传递至调度器，需要三个参数：类型、数据关键字（Key）、数据值（Value），也可以只传递类型。

行为创建器是一个单例，绑定调度器，仅仅提供调用接口，内部没有任何业务逻辑，属于中间层，用于解耦视图与调度器。

```
    private static ActionsCreator sInstance;  // 单例
    private final Dispatcher mDispatcher; // 调度器
    private ActionsCreator(Dispatcher dispatcher) {
        mDispatcher = dispatcher;
    }

    // 行为创建器单例
    public static ActionsCreator getInstance(Dispatcher dispatcher) {
        if (sInstance == null) {
            sInstance = new ActionsCreator(dispatcher);
        }
        return sInstance;
    }
```

行为创建器提供若干接口，用于视图调用，不同的接口创建不同类型的事件，有些直接传

递类型参数，如撤销销毁项（undoDestroy）、全部转换状态（toggleCompleteAll）、销毁全部选中（destroyCompleted）等；有些还要添加额外的数据字典，如创建项（create）、销毁项（destroy）、转换状态（toggleComplete）等。

```
    // 创建项
    public void create(String text) {
        mDispatcher.dispatch(TodoActions.TODO_CREATE, TodoActions.KEY_TEXT, text);
    }

    // 销毁项
    public void destroy(long id) {
        mDispatcher.dispatch(TodoActions.TODO_DESTROY, TodoActions.KEY_ID, id);
    }

    // 撤销销毁项
    public void undoDestroy() {
        mDispatcher.dispatch(TodoActions.TODO_UNDO_DESTROY);
    }

    // 转换状态
    public void toggleComplete(Todo todo) {
        long id = todo.getId();
        String actionType = todo.isComplete() ? TodoActions.TODO_UNDO_COMPLETE :
TodoActions.TODO_COMPLETE;
        mDispatcher.dispatch(actionType, TodoActions.KEY_ID, id);
    }

    // 全部转换状态
    public void toggleCompleteAll() {
        mDispatcher.dispatch(TodoActions.TODO_TOGGLE_COMPLETE_ALL);
    }

    // 销毁全部选中
    public void destroyCompleted() {
        mDispatcher.dispatch(TodoActions.TODO_DESTROY_COMPLETED);
    }
```

行为创建器仅仅起到接口（代理）的作用，解耦视图与调度器，避免因修改调度器而修改视图，解耦总线与视图逻辑。

3.2.3 调度器

调度器是一个通用类，事件分发的核心，实现事件总线（EventBus）的功能，视图与存储器双向发送。存储器端将行为创建器传递的信息封装成不同类型的行为，发送至事件总线，由存储器接收并处理；视图端将存储器状态改变的信息发送至事件总线，由视图接收并处理，更

新页面显示。

调度器是事件总线的代理，单例模式，持有事件总线变量。调度器的三个接口：register()、unregister()、post()，分别代理事件总线的三个同名接口。emitChange()是 post()的暴露接口，当存储器的数据发生变化时，调用 emitChange()，通知视图更新显示数据。

```
    private final Bus mBus;  // 事件总线
    private static Dispatcher sInstance;  // 单例

    private Dispatcher(Bus bus) {
        mBus = bus;
    }

    public static Dispatcher getInstance(Bus bus) {
        if (sInstance == null) {
            sInstance = new Dispatcher(bus);
        }
        return sInstance;
    }

    public void register(final Object cls) {
        mBus.register(cls);
    }
    public void unregister(final Object cls) {
        mBus.unregister(cls);
    }

    private void post(final Object event) {
        mBus.post(event);
    }

    // 每个状态改变都需要发送事件，由视图相应地做出更改
    public void emitChange(Store.StoreChangeEvent o) {
        post(o);

    }
```

调度器的调度接口 dispatch()，负责将事件类型、事件数据字典封装为行为，通过总线的发送接口 post()通知存储器，根据类型做出相应的数据操作。

```
    public void dispatch(String type, Object... data) {
        if (type == null || type.isEmpty()) { // 数据空
            throw new IllegalArgumentException("Type must not be empty");
        }
        if (data.length % 2 != 0) { // 非 Key-Value
            throw new IllegalArgumentException("Data must be a valid list of key,value
pairs");
        }
```

```
    Action.Builder actionBuilder = new Action.Builder().with(type);

    int i = 0;
    while (i < data.length) {
        String key = (String) data[i++];
        Object value = data[i++];
        actionBuilder.bundle(key, value); // 放置键值
    }

    // 发送到 EventBus
    post(actionBuilder.build());
}
```

调度器的两个核心接口：emitChange()和 dispatch()。emitChange()被存储器调用，通过总线通知视图，更新存储器中的显示数据；dispatch()被视图调用，通过总线通知存储器，根据行为类型操作数据。因此，调度器（总线）是连接视图（视图逻辑）和存储器（业务逻辑）的桥梁，解耦视图与存储器，两者不相互依赖，支持独立修改。

行为类非常简单，仅仅存储行为的类型和行为的数据字典。

```
private final String mType; // 类型
private final HashMap<String, Object> mData;  // 数据字典
public Action(String type, HashMap<String, Object> data) {
    mType = type;
    mData = data;
}

public String getType() {
    return mType;
}

public HashMap<String, Object> getData() {
    return mData;
}
```

行为类的构造器使用 Builder 模式，with()用于添加类型，bundle()用于添加数据对，build()用于创建行为类。行为类通过 Builder 模式构建，支持动态地添加数据对。

```
public static class Builder {
    private String mType; // 类型
    private HashMap<String, Object> mData; // 数据

    // 通过类型创建 Builder
    public Builder with(String type) {
        if (type == null) {
            throw new IllegalArgumentException("Type may not be null.");
        }
        mType = type;
```

```
            mData = new HashMap<>();
            return this;
        }

        // 绑定数据
        public Builder bundle(String key, Object value) {
            if (key == null || value == null) {
                throw new IllegalArgumentException("Key or value may not be null.");
            }
            mData.put(key, value);
            return this;
        }

        // 通过 Builder 创建 Action
        public Action build() {
            if (mType == null || mType.isEmpty()) {
                throw new IllegalArgumentException("At least one key is required.");
            }
            return new Action(mType, mData);
        }
    }
```

3.2.4　存储器

存储器用于存储或修改数据和状态，接收调度器中事件总线的修改通知，根据行为的类型，执行相应修改数据和状态的操作。数据源支持使用数据库或本地存储。

存储器都是继承于存储基类，在抽象基类中，持有调度器，并且含有存储器的对外访问接口。其中，emitStoreChange()用于发送数据修改的通知；changeEvent()用于创建修改事件，属于抽象接口，由子类覆写；onAction()用于接收调度器中事件总线的修改通知，根据不同的行为，执行相应的数据修改；StoreChangeEvent 属于数据修改事件接口，在子类中继承接口类，通过changeEvent()创建数据修改事件，发送事件至调度器中事件总线，视图类接收事件并处理。

```
    public abstract class Store {
        private final Dispatcher mDispatcher; // 调度器
        protected Store(Dispatcher dispatcher) {
            mDispatcher = dispatcher;
        }
        // 通知改变
        public void emitStoreChange() {
            mDispatcher.emitChange(changeEvent());
        }
        // 改变事件，由子类重写
        public abstract StoreChangeEvent changeEvent();
```

```
    @SuppressWarnings("unused")
    public abstract void onAction(Action action);

    // 存储改变事件
    public interface StoreChangeEvent {
    }
}
```

存储类 TodoStore 继承于基类 Store，单例(SingleInstance)模式，持有事项数据列表 mTodos、最后删除数据 lastDeleted，用于支持撤销删除功能。在构造器中，绑定调度器的原因是需要发送数据修改事件，逻辑位于基类中。getTodos()用于获取全部的事件数据；canUndo()用于判断是否可以撤销，即 lastDeleted 是否为空。

```
public class TodoStore extends Store {
    private static TodoStore sInstance; // 单例
    private final List<Todo> mTodos; // 数据列表
    private Todo lastDeleted; // 最近一次删除数据

    private TodoStore(Dispatcher dispatcher) {
        super(dispatcher);
        mTodos = new ArrayList<>();
    }
    public static TodoStore getInstance(Dispatcher dispatcher) {
        if (sInstance == null) {
            sInstance = new TodoStore(dispatcher);
        }
        return sInstance;
    }

    // 获取数据
    public List<Todo> getTodos() {
        return mTodos;
    }

    // 恢复
    public boolean canUndo() {
        return lastDeleted != null;
    }
```

onAction()是存储器的核心方法，用于处理在调度器中接收到通知，即行为。根据行为的类型不同，执行对应的逻辑，即修改数据，完成之后，向调度器发送数据已修改的广播，被视图接受，并更新页面展示。TODO_CREATE 用于根据内容创建事项，TODO_DESTROY 用于根据 ID 删除事项，TODO_UNDO_DESTROY 用于撤销已删除事项，TODO_COMPLETE 用于根据 ID 切换事项的完成状态，TODO_UNDO_COMPLETE 用于根据 ID 切换事项的非完成状态，TODO_DESTROY_COMPLETED 用于删除全部已完成状态的事项，

TODO_TOGGLE_COMPLETE_ALL 用于反转全部事项的状态。无论哪种修改逻辑，最终都需要修改数据源，因此都需要执行 emitStoreChange()，存储器向调度器发送存储状态改变的通知，再由订阅通知的视图执行页面更新操作。

```
@Override @Subscribe
public void onAction(Action action) {
    long id;  // 待处理 ID
    switch (action.getType()) {
        case TodoActions.TODO_CREATE:
            String text = ((String) action.getData().get(TodoActions.KEY_TEXT));
            create(text);
            emitStoreChange(); // 发生改变事件
            break;

        case TodoActions.TODO_DESTROY:
            id = ((long) action.getData().get(TodoActions.KEY_ID));
            destroy(id);
            emitStoreChange();
            break;

        case TodoActions.TODO_UNDO_DESTROY:
            undoDestroy();
            emitStoreChange();
            break;

        case TodoActions.TODO_COMPLETE:
            id = ((long) action.getData().get(TodoActions.KEY_ID));
            updateComplete(id, true);
            emitStoreChange();
            break;

        case TodoActions.TODO_UNDO_COMPLETE:
            id = ((long) action.getData().get(TodoActions.KEY_ID));
            updateComplete(id, false);
            emitStoreChange();
            break;

        case TodoActions.TODO_DESTROY_COMPLETED:
            destroyCompleted();
            emitStoreChange();
            break;

        case TodoActions.TODO_TOGGLE_COMPLETE_ALL:
            updateCompleteAll();
            emitStoreChange();
            break;
    }
}
```

操作示例如图 3-9 所示。

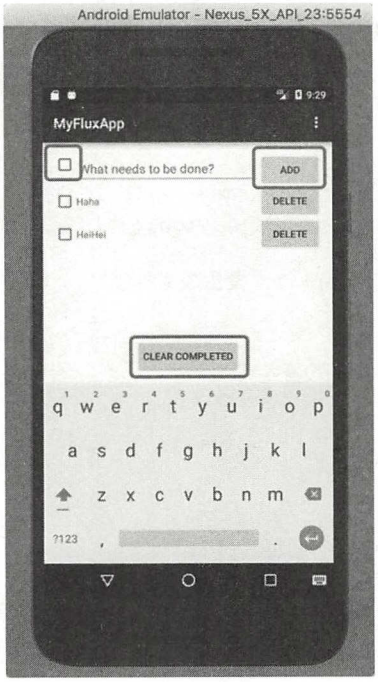


图 3-9　操作示例

接着，所列举的是修改数据的全部私有方法。

◎ destroyCompleted()：删除全部完成事项。即遍历全部事项，如果状态为完成，则删除事项。

◎ updateCompleteAll()：反转全部事项的状态。即判断是否全部完成，如果全部完成，则将事项状态全部修改为未完成；如果非全部完成，则全部修改为完成。

◎ areAllComplete()：判断全部事项状态是否都是完成。即遍历全部事项，如果全部完成，则返回 True；如果任意一个未完成，则返回 False。

◎ updateAllComplete()：更新全部事项的状态。即遍历全部事项，将事项的状态全部修改为 True 或 False。

◎ updateComplete()：更新事项的完成状态。即根据事项 ID，更新这个事项的状态，完成或未完成。

```
// 删除全部完成事项
private void destroyCompleted() {
    Iterator<Todo> iter = mTodos.iterator();
    while (iter.hasNext()) {
```

```
            Todo todo = iter.next();
            if (todo.isComplete()) {
                iter.remove();
            }
        }
    }

    // 反转全部事项的状态
    private void updateCompleteAll() {
        if (areAllComplete()) {
            updateAllComplete(false);
        } else {
            updateAllComplete(true);
        }
    }

    // 判断是否全部选中完成
    private boolean areAllComplete() {
        for (Todo todo : mTodos) {
            if (!todo.isComplete()) {
                return false;
            }
        }
        return true;
    }

    // 更新全部状态
    private void updateAllComplete(boolean complete) {
        for (Todo todo : mTodos) {
            todo.setComplete(complete);
        }
    }

    // 更新 ID 的完成状态
    private void updateComplete(long id, boolean complete) {
        Todo todo = getById(id);
        if (todo != null) {
            todo.setComplete(complete);
        }
    }
```

◎ undoDestroy()：撤销被删除的事项。即判断是否含有删除事项，如果含有，则重新添加上一次被删除的事项 lastDeleted。

◎ create()：根据内容创建事项，重新排序事项列表。即将当期系统时间设置为事项 ID，通过 ID 和内容创建事项，加入事项列表，重新排序列表。

◎ destroy()：删除事项，保存上一次被删除事项（lastDeleted）。即根据事项 ID，将

待删除的事项保存至上一次被删除事项（，用于撤销，接着删除事项。

◎ getById()：根据事项 ID 获取事项。即遍历事项列表，获取待查询 ID 的事项。

◎ addElement()：添加事项。即将事项添加至事项列表，重新排序列表。

```
// 撤销删除的事项
private void undoDestroy() {
    if (lastDeleted != null) {
        addElement(lastDeleted.clone());
        lastDeleted = null;
    }
}

// 根据 Text 创建事项，并且排序
private void create(String text) {
    long id = System.currentTimeMillis();
    Todo todo = new Todo(id, text);
    addElement(todo);
    Collections.sort(mTodos);
}

// 根据 ID 删除事项，并且放入最后删除项
private void destroy(long id) {
    Iterator<Todo> iter = mTodos.iterator();
    while (iter.hasNext()) {
        Todo todo = iter.next();
        if (todo.getId() == id) {
            lastDeleted = todo.clone();
            iter.remove();
            break;
        }
    }
}

// 获取根据 ID 获取事项
private Todo getById(long id) {
    Iterator<Todo> iter = mTodos.iterator();
    while (iter.hasNext()) {
        Todo todo = iter.next();
        if (todo.getId() == id) {
            return todo;
        }
    }
    return null;
}

// 添加数据进入列表
private void addElement(Todo clone) {
```

```
        mTodos.add(clone);
        Collections.sort(mTodos);
    }
```

最后，实现基类的抽象接口 changeEvent()，用于创建存储事件；存储事件 TodoStoreChangeEvent 继承于抽象类的接口 StoreChangeEvent()。更丰富、更复杂的信息放置于 TodoStoreChangeEvent 类中，传递至视图解析。本例作为讲解，逻辑简单，并未编写 TodoStoreChangeEvent 类中的逻辑。

```
@Override
public StoreChangeEvent changeEvent() {
    return new TodoStoreChangeEvent();
}

public class TodoStoreChangeEvent implements StoreChangeEvent {
}
```

其余事项列表 RecyclerView 的逻辑类 RecyclerAdapter 也属于视图（View），与 MainActivity 类似；事项类 Todo 持有事项 ID、事项内容、事项完成状态，提供访问和修改接口，覆写比较函数。RecyclerAdapter 类和 Todo 类不作详细讲解。至此，基于 Flux 架构的 Android 工程，其基本框架和要点已经全部介绍完毕，四大核心组件：视图、行为创建器、调度器、存储器都需要详细掌握。

- ◎ 视图：负责接收用户事件和展示页面数据。通过回调接口接收用户事件，将用户事件封装成行为，通过调度器发送行为。通过总线接口接收数据行为，当数据修改时，更新数据源，重新展示页面。
- ◎ 行为创建器：将视图和存储器统一封装成行为，行为含有类型和数据，当接收到行为时，根据行为的类型执行相应的操作。
- ◎ 调度器：属于总线系统，连接视图和存储器，传递被行为创建器封装的行为，将行为发送至订阅行为的类，属于观察者模式。
- ◎ 存储器：负责响应用户事件和通知页面修改。从调度器中接收从视图传递来的行为，根据行为的类型不同，相应地修改数据源，当修改完成后，向调度器发送数据修改完成的通知。

Flux 架构的核心就是基于观察者模式，即总线系统，连接存储器与视图，接收用户事件，反馈数据修改，将业务逻辑与视图逻辑解耦。Flux 架构的另一个显著特点在于数据的更新，对于简单的视图全量更新数据即可，而对于复杂的视图则可以定点更新。总而言之，Flux 架构比较灵活，通过总线解耦各个模块之间的依赖。在开发项目中，理解 Flux 架构的思想，不仅可以应用于工程级别的设计，也可以应用于模块级别的设计，思想大同小异，核心在于解耦。

想更多的了解 Flux 架构，参考 Facebookhttps://facebook.github.io/flux/docs/overview.html。

第 4 章
响应式编程

4.1 全面解析响应式库 RxJava 的使用方式

RxJava 是一个响应式编程框架，实现异步操作，如事件响应、网络请求等，如图 4-1 所示。R 表示 Reactive（响应式），x 表示任何，则 Rx 表示将响应式的编程思想应用于任何语言，如 Java，因此 RxJava 就是响应式编程的 Java 版本。同时，在 RxJava 的基础上，还扩展成为 RxAndroid，引入了 Android 系统的线程概念。

响应式编程的核心思想就是观察者模式（设计模式）。对于观察者模式，其需求本质是：对象 A（观察者）高度敏感地关注对象 B（被观察者）的变化，当对象 B 发生变化的瞬间，对象 A 给予适当的反应（响应）。观察者模式分为两种，即主动模式与被动模式。主动模式就是观察者实时监控被观察者的状态，根据被观察者的状态变化，观察者做出反馈。被动模式就是观察者在被观察者中订阅或注册，当被观察者发生状态变化时，通知观察者，观察者做出反馈。主动模式以观察者为主导，被动模式以被观察者为主导。

举例来说，生活中警察与小偷的关系就是观察者与被观察者的关系。将警察与小偷绑定后，警察时刻在盯着小偷的一举一动，当小偷伸手作案时（状态变化），警察实施抓捕（响应），这种观察者模式属于主动模式。再比如 Android 系统中的监听器（Listener）与控件的关系，也是观察者与被观察者的关系。将监听器添加在控件中（订阅），当控件接收到用户行为时（状态变化），监听器执行接口中的逻辑（响应），这种观察者模式属于被动模式。

RxJava 采用观察者模式的被动模式，通过订阅的方式将观察者与被观察者绑定，由观察者监控被观察者的状态，当被观察者的状态发生变化时，通知观察者做出适当的反应。

图 4-1　RxJava

在理解了 RxJava 的原理后，又应该如何运用 RxJava，也就是采用响应式编程开发 Android 项目呢？这是本章关注的重点。在开发过程中，开发人员应学习 RxJava 的高级知识，灵活运行并融会贯通。本章主要介绍 RxJava 的五种基本用法：

（1）掌握 RxJava 响应式编程的链式表达式；

（2）熟悉 RxJava 常见流的加工函数，方便地管理异步数据流；

（3）使用 RxJava & Lambda 表达式，避免冗余代码；

（4）使用 RxJava & Retrofit，处理异步的网络请求；

（5）使用 RxJava & RxBinding，处理异步的控件事件；

（6）使用 RxJava & RxLifecycle，管理线程的循环任务，避免内存泄漏。

本章实例完整代码的下载地址：https://github.com/SpikeKing/TestDetailRxAndroid

4.1.1　项目框架

本节从一个新的空白项目开始，以 Empty Activity 为主页面，从零开始搭建 RxJava 的相关框架，便于不熟悉 Android 项目构建的读者加深理解。

对于一个 Android 工程而言，有两类 Gradle 配置文件，一类是项目的 Gradle 配置，一类是工程的 Gradle 配置。每个项目（或称为模块）都有属于自己的 Gradle 配置，整个工程有一个通用的 Gradle 配置。项目的 Gradle 配置主要用于管理项目的属性与依赖；工程的 Gradle 配置主要用于管理全部项目的通用属性和工程的构建，如工程构建脚本（buildscript）、库的下载地址（jcenter）等。

比如，工程最基本的 Gradle 配置包含三个部分，即 buildscript、allprojects、task clean。其中，buildscript 表示构建脚本，依赖库来源于 Jcenter，依赖于 Gradle 的 Build Tools 工具包；allprojects 表示全部项目的通用配置，依赖库来源于 Jcenter；task clean 表示清除命令（gradle clean）脚本，删除工程根目录的 build 文件夹。

注意：JCenter 与 Maven Central 都是下载第三方依赖库的仓库。相比于 Maven Central，JCenter 拥有更多的依赖库，下载性能也更好，是 Gradle 的内置仓库。但是 Maven Central 不是 JCenter 的子集，某些特定的依赖库仅存在于 Maven Central 中。

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:2.3.2'
    }
}

allprojects {
    repositories {
        jcenter()
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
```

当开发小型工程时，开发人员很少会修改工程的 Gradle 配置，而是更关注项目的 Gradle 配置。在 Gradle 中，一个重要的部分就是添加第三方库的依赖（dependencies）。使用第三方库，可以避免重复创建轮子，加快开发速度，提升代码的稳定性。本例主要关注 RxJava 的使用，为了让读者更全面地理解其广泛的用途，引入若干相关联的库，如：

（1）ButterKnife：布局 Id 的依赖注入库，减少编写查找布局 Id 的代码；

（2）RxJava 与 RxAndroid：核心库，两者配合使用，RxJava 提供基础的响应式编程框架，RxAndroid 提供针对 Android 系统的补充；

（3）Retrofit：网络加载库，封装 OKHttp 库，简洁地编写网络请求；

（4）Picasso：图片加载库，二级缓存，快速网络或本地图片；

（5）RxLifecycle：Rx 线程的管理库，自动管理 Rx 线程的释放；

（6）RxBinding：控件的异步监听库，使用响应式编程模式设置控件监听。

这些第三方库的具体使用方式，将通过本例的实例代码进行讲解，这里只做一些简要的说明。

```
compile 'com.jakewharton:butterknife:7.0.1' // ButterKnife 标注
```

```
// RxAndroid
compile 'io.reactivex:rxandroid:1.1.0'
compile 'io.reactivex:rxjava:1.1.0' // 推荐同时加载 RxJava

// Retrofit 网络处理
compile 'com.squareup.retrofit:retrofit:2.0.0-beta2'
compile 'com.squareup.retrofit:adapter-rxjava:2.0.0-beta2'
compile 'com.squareup.retrofit:converter-gson:2.0.0-beta2'

// Picasso 网络图片加载
compile 'com.squareup.picasso:picasso:2.5.2'

// RxLifecycle 管理 Rx 的生命周期
compile 'com.trello:rxlifecycle:0.4.0'
compile 'com.trello:rxlifecycle-components:0.4.0'

// RxBinding
compile 'com.jakewharton.rxbinding:rxbinding:0.3.0'
compile 'com.jakewharton.rxbinding:rxbinding-appcompat-v7:0.3.0'
compile 'com.jakewharton.rxbinding:rxbinding-design:0.3.0'
```

除了依赖库，为了更好地配合 RxJava 的响应式编程，本例引入 Lambda 表达式。Lambda 表达式是一个匿名函数，因数学中的 Lambda 演算而得名，直接对应于其中的 Lambda 抽象，则是一个匿名函数，即没有函数名的函数。通过 Lambda 表达式，省略多层嵌套的匿名类和匿名接口，改写为一层 Lambda 表达式，仅含有输入、输出、逻辑三个部分，增强代码的可读性，使编写的代码更加优雅。本例使用 Tatarka 的 retrolambda 库插件（plugins），同时引入 Java 1.8 的编译版本（retrolambda）。

```
plugins {
    id "me.tatarka.retrolambda" version "3.2.4"
}

android {
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
}
```

其他关于项目的 Gradle 配置，仅做少量修改，具体参考工程的源码。

4.1.2　链式表达式

首先修改主页 MainActivity.java，添加从主页到其他页面的跳转，直接使用 startActivity() 的方式。跳转接口与页面的布局联动，在 activity_main.xml 中，设置控件 Button 的 android:onClick

属性，表明跳转接口名称 gotoSimpleModule；跳转接口位置，位于顶层布局 LinearLayout 的 tools:context 属性，表明跳转文件名称 MainActivity。

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context="org.wangchenlong.testdetailrxandroid.MainActivity">

    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:onClick="gotoSimpleModule"
        android:text="@string/test_button_simple"/>
```

在主页 MainActivity 中，提供 6 种跳转接口，分别对应 6 种不同的页面。gotoSimpleModule 表示 RxJava 最基础的使用方式，类似于编程中的“HelloWorld”；gotoMoreModule 表示 RxJava 稍复杂的使用方式，含有一些常用的异步处理逻辑；gotoLambdaModule 表示 RxJava 的 Lambda 版本使用方式，通过 Lambda 表达式，更简洁地替换匿名类，更直观地展示逻辑；gotoNetworkModule 表示在网络层 RxJava 的使用方式，异步的网络请求是 RxJava 最常用的地方之一；gotoSafeModule 表示 RxJava 线程安全的使用方式，异步请求可能会导致内存泄漏，通过 RxJava 的生命周期类可以优雅地避免此类情况；gotoBindingModule 表示 RxJava 的 RxBinding 形式使用方式，使用 RxBinding 绑定异步回调事件，异步回调事件也是 RxJava 常用的地方之一。

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void gotoSimpleModule(View view) {
        startActivity(new Intent(this, SimpleActivity.class));
    }

    public void gotoMoreModule(View view) {
        startActivity(new Intent(this, MoreActivity.class));
    }

    public void gotoLambdaModule(View view) {
        startActivity(new Intent(this, LambdaActivity.class));
    }

    public void gotoNetworkModule(View view) {
        startActivity(new Intent(this, NetworkActivity.class));
    }
```

```
    public void gotoSafeModule(View view) {
        startActivity(new Intent(this, SafeActivity.class));
    }

    public void gotoBindingModule(View view) {
        startActivity(new Intent(this, BindingActivity.class));
    }
}
```

接着，在 SimpleActivity 中，来学习 RxJava 最基础的使用方式，这是纯粹地讲解 RxJava 的使用方式，与实际应用无关，因为先学会基础的使用方式，才能应用于实际开发。在 onCreate() 中，RxJava 的实现由三部分构成，第一部分注册观察活动，第二部分设置观察线程，第三部分注册订阅者（观察者），属于非常标准的观察者模式。观察活动产生异步数据，当完成获取数据时，通知在观察线程中注册的订阅者，并把数据分发至订阅者的接口，执行后续的处理逻辑。

```
@Override protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_simple);
    ButterKnife.bind(this);

    // 注册观察活动
    Observable<String> observable = Observable.create(mObservableAction);

    // 设置观察线程
    observable.observeOn(AndroidSchedulers.mainThread());

    // 分发订阅信息
    observable.subscribe(mTextSubscriber);
    observable.subscribe(mToastSubscriber);
}
```

在注册观察活动中，调用 Observable 的 create()方法，参数是 mObservableAction，输出 Observable<String>类，表明观察活动所产生的异步数据是 String 对象类型。mObservableAction 类是订阅接口类，为观察活动提供数据源，并调用 call()接口，执行被注册订阅者的 onNext()与 onCompleted()方法。本部分的数据源较为简单，就是一个返回 String 对象类型的方法，属于同步方法，此处也可以更换为异步方法。

```
// 观察事件发生
private Observable.OnSubscribe<String> mObservableAction = new Observable.
OnSubscribe<String>() {
    @Override public void call(Subscriber<? super String> subscriber) {
        subscriber.onNext(sayMyName()); // 发送事件
        subscriber.onCompleted(); // 完成事件
    }
};
```

```
// 创建字符串
private String sayMyName() {
    return "Hello, I am your friend, Spike!";
}
```

在设置观察线程中，调用 Observable 的 observeOn()方法，参数是 AndroidSchedulers.mainThread()，表明观察者在哪个线程中执行。一般而言，如果观察者需要更新界面，则必须位于主线程（UI 线程）；如果订阅者需要执行网络请求，则必须位于子线程。AndroidSchedulers 就是RxAndroid相比于RxJava的最大不同之处，它额外提供了一个基于Android系统的主线程。除此之外，在 RxJava 的 Schedulers 类中，还提供多种其他线程，如 computation()计算线程、io()I/O 线程、newThread()新的线程、test()测试线程、immediate()当前线程（立即执行）、trampoline()当前线程（队列执行）、from()由执行器转变的线程等，根据不同的情况，选择最合适的线程执行，如图 4-2 所示。

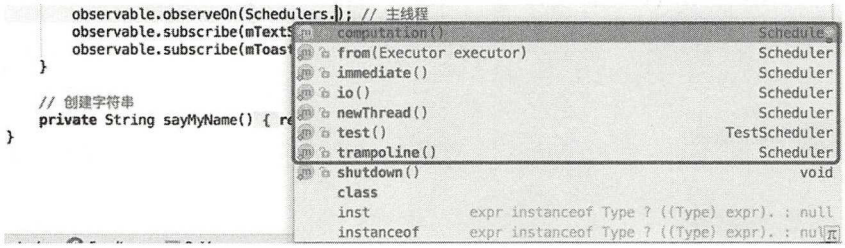


图 4-2　Schedulers 的其他线程

在注册订阅者中，调用 Observable 的 subscribe()方法，参数是 Subscriber<String>，类型与观察活动保持一致，都是 String 类型。本例声明两个订阅者，一个用于设置 TextView 控件的文字，一个用于输出 Toast 提示。每个订阅者都含有 onNext、onCompleted、onError 三种方法，分别表示后续操作、完成操作和错误操作。与输入输出流最为相关的是 onNext 方法，参数就是在观察活动中传递的数据；onError 主要处理异步过程中产生的异常。

```
// 订阅者，接收字符串，修改控件
private Subscriber<String> mTextSubscriber = new Subscriber<String>() {
    @Override public void onCompleted() {
    }

    @Override public void onError(Throwable e) {
    }

    @Override public void onNext(String s) {
        mTvText.setText(s); // 设置文字
    }
};
```

```
// 订阅者，接收字符串，提示信息
private Subscriber<String> mToastSubscriber = new Subscriber<String>() {
    @Override public void onCompleted() {
    }

    @Override public void onError(Throwable e) {
    }

    @Override public void onNext(String s) {
        Toast.makeText(SimpleActivity.this, s, Toast.LENGTH_SHORT).show();
    }
};
```

由此，最基础的 RxJava 使用方式已经讲解完成，如图 4-3 所示。其核心思想就是观察者模式，设置被观察者的对象，选择操作执行的线程，设置订阅者的对象，并把三者串联起来。

图 4-3　基础

4.1.3　流的加工函数

上一节已经初步介绍了 RxJava 的链式表达式，但这种使用方式比较冗长，与拖沓的 Java 编程方式有关，因此 RxJava 提供了多种封装好的方法，可以直接使用封装好的函数输出想要的结果。比如，一些常用的方法如下：

（1）**Just**。输入待分发的异步接口，调用完成后直接分发，使编写更加简洁，省略其他回调。

（2）**From**。与“just”类似，异步接口的数据变成数组，将数组转变成单个元素，分发多次。

（3）**map**。映射，调用大小写转换、控制取值范围等特定的方法加工数据流，继续分发。

（4）**flatMap**。增肥映射，“flat”的本意是增肥，与“map”类似，“flatMap”映射之后返回的是观察者对象，而“map”映射之后返回的是数值，都是修改数据流，继续分发。

（5）**reduce**。简化，把数组的多个值合成一个数据值，分发一次。

下面通过实例，来介绍如何使用这些流的加工函数。在本例中，会引入两个 Action 和三个 Func（Function）。Action 类似于订阅者（消费者），订阅最后的数据，输出可视化结果，一个在 TextView 中显示，另一个在 Toast 中显示。Action 后面的数字表示可以“消费”的参数有几个，Action1 表示可以“消费”一个参数。Func 类似于加工函数，第一个 Func 将数组中的元素分别分发，第二个 Func 将 String 中的字母转换为大写字母，第三个 Func 将两个字符串合并为一个字符串。Func 后面的数字表示可以“加工”的参数有几个，Func2 表示可以“加工”两个参数，但是 Func 的实际模板参数比加工参数多一个，最后一个表示需要返回的参数类型。

```
final String[] mManyWords = {"Hello", "I", "am", "your", "friend", "Spike"};
final List<String> mManyWordList = Arrays.asList(mManyWords);

// Action 类似于订阅者,设置 TextView
private Action1<String> mTextViewAction =
        new Action1<String>() {
            @Override public void call(String s) {
                mTvText.setText(s);
            }
        };

// 设置 Toast
private Action1<String> mToastAction = new Action1<String>() {
    @Override public void call(String s) {
        Toast.makeText(MoreActivity.this, s, Toast.LENGTH_SHORT).show();
    }
};

// 设置映射函数
private Func1<List<String>, Observable<String>> mOneWordFunc =
        new Func1<List<String>, Observable<String>>() {
            @Override public Observable<String> call(List<String> strings) {
                return Observable.from(strings); // 映射字符串
            }
        };
// 设置大写字母
private Func1<String, String> mUpperLetterFunc =
        new Func1<String, String>() {
            @Override public String call(String s) {
                return s.toUpperCase(); // 大写字母
            }
        };

// 连接字符串
private Func2<String, String, String> mMergeStringFunc =
        new Func2<String, String, String>() {
```

```
            @Override public String call(String s, String s2) {
                return String.format("%s %s", s, s2); // 空格连接字符串
            }
        };

// 创建字符串
private static String sayMyName() {
    return "Hello, I am your friend, Spike!";
}
```

接下来，在 RxJava 中使用这些 Action 和 Func 组成异步的数据加工流。

第一段代码的逻辑。首先，通过 just 方法，创建一个被观察者对象 Observable，数据流的值就是 sayMyName()函数的返回值；其次，设置观察线程是 Android 的 UI 线程；然后，调用加工函数 map，将数据流的字母全部转变为大写字母；最后，输出到订阅者 mTextViewAction 中，在 TextView 中，显示最终的 String 字符串。当然，两部分代码也可以合并在一起使用，例如：just -> observeOn -> map -> subscribe，会更加简洁。这次简化了订阅者的逻辑，并未处理异常，仅提供数据流的正常处理逻辑。

```
// 添加字符串，省略 Action 的其他方法，只使用一个 onNext.
Observable<String> obShow = Observable.just(sayMyName());

// 先映射，再设置 TextView
obShow.observeOn(AndroidSchedulers.mainThread())
        .map(mUpperLetterFunc).subscribe(mTextViewAction);
```

第二段代码的逻辑。首先，通过 from 方法将数组 mManyWords 中的数据多次分发，每次分发一个元素；其次，同样设置观察线程是 Android 的 UI 线程；然后，调用加工函数 map，将字符串转换为大写字母的字符串；最后，输出到订阅者 mToastAction 中。在 Toast 中，连续提示多个字符串。

```
// 单独显示数组中的每个元素
Observable<String> obMap = Observable.from(mManyWords);

// 映射之后分发
obMap.observeOn(AndroidSchedulers.mainThread())
        .map(mUpperLetterFunc).subscribe(mToastAction);
```

第三段代码的逻辑。首先，通过 just 方法，将整个数组放入数据流中一起分发；其次，同样设置观察线程是 Android 的 UI 线程；然后，调用 flatMap 方法，将数组的每个元素转变为一个被观察对象，独立分发；接着，调用 reduce 方法，将多个观察者的数据合并成一个数据；最后，输出到订阅者 mToastAction 中，在 Toast 中，只输出一个字符串。

```
// 优化过的代码，直接获取数组,再分发、合并、显示 toast，Toast 顺次执行.
Observable.just(mManyWordList)
        .observeOn(AndroidSchedulers.mainThread())
```

```
        .flatMap(mOneWordFunc)
        .reduce(mMergeStringFunc)
        .subscribe(mToastAction);
```

由此可以发现，在这三段逻辑中，RxJava 的写法有多种多样，调用合适的方法解决特定的问题，使得整个逻辑优雅、简单和易读，如图 4-4 所示。

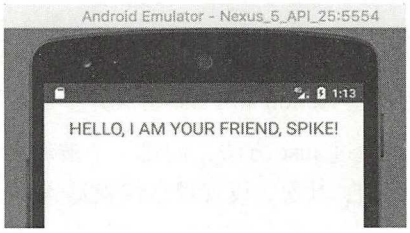


图 4-4　加工方法

4.1.4　Ambda 表达式

Lambda 表达式（Lambda Expression）是一种匿名函数，与数学中的 λ 演算类似，对应演算中的 Lambda 抽象，没有名称的函数。Java 是对象式编程语言，遵循对象至上的原则。在基本的编程规范中，函数无法独立于对象存在，而函数式编程语言提供了一种强大的功能——闭包。相比于传统的编程语言，闭包具有很多优势，使得函数可以独立于对象存在。闭包类似一个可调用的对象，可以记录作用域的相关信息。因此，Java 1.8 使用 Lambda 表达式代替匿名类的方法，其概念接近于闭包。

在 RxJava 中，流的每一个过程都是一个匿名类（如 Func 或 Action），调用其中的函数，使用 Java 的 Lambda 表达式，使得整个流程简洁、易读。因为输入与输出在流中很容易观察，不会影响代码的可读性，Lambda 表达式大幅减少缩进，使得 RxJava 的编写更加流畅。

本节把上一个示例重新用 Java 的 Lambda 表达式改写，保持功能相同。读者可以领略 Lambda 表达式的优雅之处。

```
// 添加字符串，省略 Action 的其他方法，只使用一个 onNext
Observable<String> obShow = Observable.just(sayMyName());

// 先映射，再设置 TextView
obShow.observeOn(AndroidSchedulers.mainThread())
        .map(String::toUpperCase).subscribe(mTvText::setText);

// 单独显示数组中的每个元素
Observable<String> obMap = Observable.from(mManyWords);

// 映射之后分发
```

```
obMap.observeOn(AndroidSchedulers.mainThread())
        .map(String::toUpperCase)
        .subscribe(this::showToast);

// 优化过的代码，直接获取数组，再分发、合并、显示 toast，Toast 顺次执行.
Observable.just(mManyWordList)
        .observeOn(AndroidSchedulers.mainThread())
        .flatMap(Observable::from)
        .reduce(this::mergeString)
        .subscribe(this::showToast);
```

这次没有使用常规的 Lambda 表达式，而是更为简单的模式——方法引用（Method References）。方法引用的含义是：当方法的参数和返回值与匿名类函数的参数和返回值相同时，不需要指明传递的参数，直接使用方法名代替即可。在方法名之前，需要添加类的名称，使用“::”表示所属关系，this 表示当前类。如下就是 showToast 方法的方法引用与 Lambda 表示式的写法比较，可见方法引用更为简洁。

```
.subscribe(this::showToast);  // 方法引用
.subscribe(s -> showToast(s));  // Lambda 表达式
```

在低版本 Java 的 Android 项目中，如果希望使用 Java 1.8 版本的 Lambda 表达式，可以集成 Retrolambda 的开源库，启动 Lambda 表达式的功能。Retrolambda 的配置非常简单，在项目的 build.gradle 文件中，添加 Retrolambda 的 plugins（插件），同时指定 Android 兼容 Java 1.8 版本即可。

```
plugins {
    id "me.tatarka.retrolambda" version "3.2.4"
}
android {
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
}
```

4.1.5　网络请求

Retrofit 是一个可用于 Android 的网络请求库，简化网络请求的编写，如 Get 请求或 Post 请求，提高开发效率。Retrofit 基于 OkHttp，在其之上进一步封装，最终的网络请求都是交给 OkHttp 处理。在 Retrofit 的外层接口，仅仅使用注解，就能实现复杂的网络请求功能，完成客户端与服务器端的数据交互，开发极其高效。RxJava 的核心功能是处理异步任务，而网络请求正是最常用的异步任务，同时，Retrofit 网络请求库也提供支持的 RxJava 接口。在实现网络请求的过程中，同时使用 Retrofit 和 RxJava，可以极大地提高开发效率。

在 Gradle 配置中，引入 Retrofit 的相关代码库。

```
compile 'com.squareup.retrofit:retrofit:2.0.0-beta2' // Retrofit
compile 'com.squareup.retrofit:adapter-rxjava:2.0.0-beta2' // RxJava 适配器
compile 'com.squareup.retrofit:converter-gson:2.0.0-beta2' // Gson 转换器
```

我们创建一个简单的列表视图，用于展示 GitHub 的用户信息，展示页面使用一个列表视图（RecyclerView），网络请求部分使用 Retrofit 和 RxJava 的组合。首先设置竖直的滚动方向（LinearLayoutManager），其次设置适配器（UserListAdapter），最后通过网络请求加载网络数据（NetworkWrapper）。注意 UserListAdapter 的参数 this::gotoDetailPage，上一小节讲过，属于方法引用，调用过程 UserListAdapter 的构造器至 UserClickCallback，再由 UserClickCallback 的 onItemClicked 调用至 gotoDetailPage，同时 onItemClicked 的输入参数与 gotoDetailPage 的输入参数相同。

```
public class NetworkActivity extends Activity {
    @Bind(R.id.network_rv_list) RecyclerView mRvList; // 列表

    @Override protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_network);
        ButterKnife.bind(this);

        // 设置 Layout 管理器
        LinearLayoutManager layoutManager = new LinearLayoutManager(this);
        layoutManager.setOrientation(LinearLayoutManager.VERTICAL);
        mRvList.setLayoutManager(layoutManager);

        // 设置适配器
        UserListAdapter adapter = new UserListAdapter(this::gotoDetailPage);
        NetworkWrapper.getUsersInto(adapter);  // 加载网络信息
        mRvList.setAdapter(adapter);
    }

    // 点击的回调
    public interface UserClickCallback {
        void onItemClicked(String name);
    }

    // 跳转到库详情页面
    private void gotoDetailPage(String name) {
        startActivity(NetworkDetailActivity.from(NetworkActivity.this, name));
    }
}
```

UserListAdapter 是 RecyclerView 列表的适配器，在构造器中，设置 UserClickCallback 回调，在 UserViewHolder 中每次点击项时调用回调；在 addUser 中暴露给外部，提供更新数据的接口，

动态添加，并且每次更新最后一个数据（notifyItemInserted）；在适配器的两个默认接口 onCreateViewHolder 和 onBindViewHolder 中，通过 UserViewHolder 绑定数据至页面；GitHubUser 是一个 JavaBean 类，负责限定展示的数据结构。其中，使用 ButterKnife 完成布局的绑定，同时使用 Picasso 完成图片控件的填充，ButterKnife 和 Picasso 都是常用的第三方控件，可提升开发效率。

```
    public class UserListAdapter extends RecyclerView.Adapter<UserListAdapter.
serViewHolder> {
        private List<GitHubUser> mUsers; // 用户名集合
        private NetworkActivity.UserClickCallback mCallback; // 用户点击项的回调

        public UserListAdapter(NetworkActivity.UserClickCallback callback) {
            mCallback = callback;
            mUsers = new ArrayList<>();
        }

        public void addUser(GitHubUser user) {
            mUsers.add(user);  // 添加数据
            notifyItemInserted(mUsers.size() - 1);  // 更新最后一位
        }

        @Override public UserViewHolder onCreateViewHolder(ViewGroup parent, int
viewType) {
            View item = LayoutInflater.from(parent.getContext())
                    .inflate(R.layout.item_network_user, parent, false);
            return new UserViewHolder(item, mCallback);
          }

        @Override public void onBindViewHolder(UserViewHolder holder, int position)
{
            holder.bindTo(mUsers.get(position));
        }

        @Override public int getItemCount() {
            return mUsers.size();
        }

        // Adapter 的 ViewHolder
        public static class UserViewHolder extends RecyclerView.ViewHolder {
            @Bind(R.id.network_item_iv_user_picture) ImageView mIvUserPicture;
            @Bind(R.id.network_item_tv_user_name) TextView mTvUserName;
            @Bind(R.id.network_item_tv_user_login) TextView mTvUserLogin;
            @Bind(R.id.network_item_tv_user_page) TextView mTvUserPage;

            public UserViewHolder(View itemView, NetworkActivity.UserClickCallback
callback) {
                super(itemView);
```

```
            ButterKnife.bind(this, itemView);
            // 绑定点击事件
            itemView.setOnClickListener(v ->
                    callback.onItemClicked(mTvUserLogin.getText().toString()));
        }

        // 绑定数据
        public void bindTo(GitHubUser user) {
            mTvUserName.setText(user.name);
            mTvUserLogin.setText(user.login);
            mTvUserPage.setText(user.repos_url);

            Picasso.with(mIvUserPicture.getContext())
                    .load(user.avatar_url)
                    .placeholder(R.drawable.ic_person_black_24dp)
                    .into(mIvUserPicture);
        }
    }
    // 用户类，名称必须与 Json 解析相同
    public static class GitHubUser {
        public String login;
        public String avatar_url;
        public String name;
        public String repos_url;
    }
}
```

自动生成 Json 解析的 JavaBean 类(如 GitHubUser)的网站：http://www.jsonschema2pojo.org/ ，同样地，Android Studio 的 GsonFormat 插件也可以实现自动生成，如图 4-5 所示。

页面的网络请求类 NetworkWrapper 负责从 GitHub 下载网络数据，并且在页面中动态地显示这些数据。NetworkWrapper 有两个接口，一个是 getUsersInto，用于下载多个 GitHub 知名用户的信息，如 SpikeKing、JakeWharton 等，同时调用 UserListAdapter 的 addUser 接口，动态地将用户信息显示在列表中；一个是 getReposInfo，用于根据用户名下载用户的仓库（Repo）信息，同时调用 RepoListAdapter 的 addRepo 接口，动态地将仓库信息显示在列表中。

在 getUsersInto 接口中，调用 Retrofit 服务 GitHubService，使用工厂模式类 ServiceFactory 的 createServiceFrom 方法创建服务，参数是类对象 GitHubService.class 和网址 GitHubService.ENDPOINT。在 RxJava 的观察者流中，from 将数组中的元素依次分发，flatMap 将元素输入至 GitHubService 的 getUserData 方法中，并输出数据，subscribe 将数据设置至 UserListAdapter 的 addUser 方法中，完成整个网络请求流，由 GitHubService 的 getUserData 方法处理异步的网络请求。

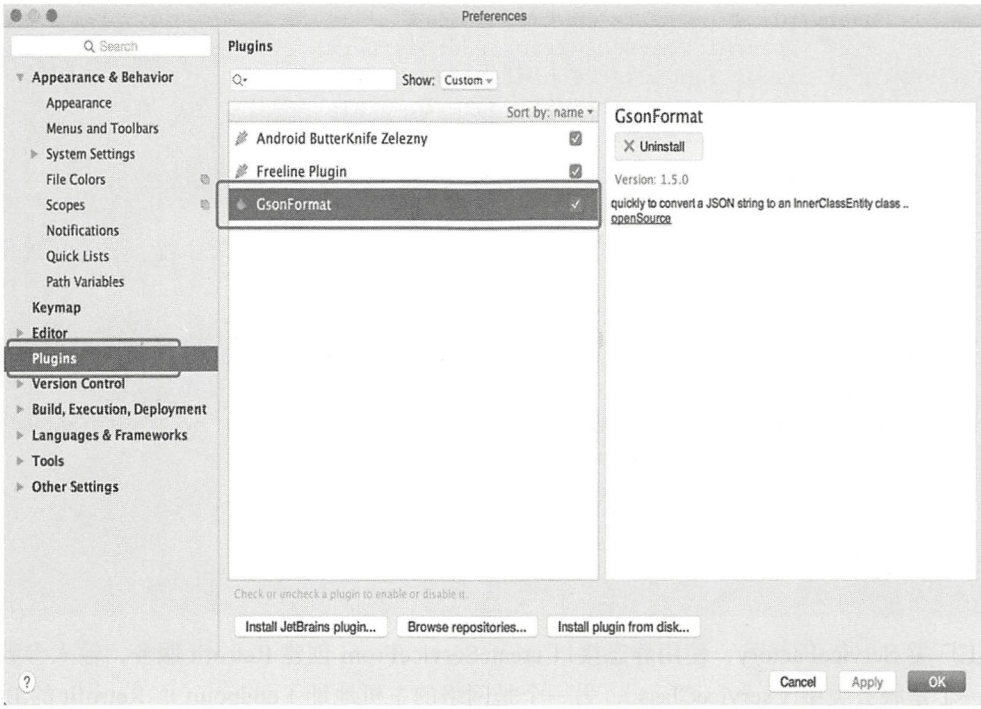


图 4-5　Gson Format

在 getReposInfo 接口中略有不同，RxJava 的观察者流起源于 GitHubService 的 getRepoData 接口，根据用户名获取用户的仓库列表，flatMap 通过调用 from 将列表中的数据依次分发，subscribe 将数据设置至 RepoListAdapter 的 addRepo 方法中，完成整个网络请求流，由 GitHubService 的 getReposInfo 方法处理异步的网络请求。

注意，网络请求属于耗时任务，影响主线程的 UI 显示，因此 Android 系统要求，网络请求无法在主线程中执行，必须在额外的线程中执行，如本例在 Schedulers.newThread()中执行，即 subscribeOn 所在的线程，否则会产生异常，导致程序崩溃。

```
    public class NetworkWrapper {
        private static final String[] mFamousUsers =
                {"SpikeKing",  "JakeWharton",  "rock3r",  "Takhion",  "dextorer",
"Mariuxtheone"};

        // 获取用户信息
        public static void getUsersInto(final UserListAdapter adapter) {
            GitHubService gitHubService =
                    ServiceFactory.createServiceFrom(GitHubService.class,
GitHubService.ENDPOINT);
```

```
            Observable.from(mFamousUsers)
                    .flatMap(gitHubService::getUserData)
                    .subscribeOn(Schedulers.newThread())
                    .observeOn(AndroidSchedulers.mainThread())
                    .subscribe(adapter::addUser);
        }

        // 获取库信息
        public static void getReposInfo(final String username, final RepoListAdapter
adapter) {
            GitHubService gitHubService =
                    ServiceFactory.createServiceFrom(GitHubService.class,
GitHubService.ENDPOINT);

            gitHubService.getRepoData(username)
                    .flatMap(Observable::from)
                    .subscribeOn(Schedulers.newThread())
                    .observeOn(AndroidSchedulers.mainThread())
                    .subscribe(adapter::addRepo);
        }
    }
```

工厂类 ServiceFactory，使用静态接口 createServiceFrom 创建 Retrofit 服务，输入参数有两个，一个是服务类型（serviceClass），另一个是网络的主机地址（endpoint）。Retrofit 的 Builder 类负责构建 Retrofit 服务，baseUrl 用于添加请求的主机地址，addCallAdapterFactory 用于添加网络流适配器，本例使用 RxJava 适配器，addConverterFactory 用于添加网络数据转换器，本例使用 Gson。最后，Builder 与服务类共同创建完整的 Retrofit，用于处理网络请求，如 Get 或 Post。

```
    public class ServiceFactory {
        public static <T> T createServiceFrom(final Class<T> serviceClass, String
endpoint) {
            Retrofit adapter = new Retrofit.Builder()
                    .baseUrl(endpoint)
                    .addCallAdapterFactory(RxJavaCallAdapterFactory.create())
                    // 添加 Rx 适配器
                    .addConverterFactory(GsonConverterFactory.create())
                    // 添加 Gson 转换器
                    .build();
            return adapter.create(serviceClass);
        }
    }
```

RxJava 适配器和 Gson 转换器都需要依赖相应的 maven 库：

```
    compile 'com.squareup.retrofit:adapter-rxjava:2.0.0-beta2'
    compile 'com.squareup.retrofit:converter-gson:2.0.0-beta2'
```

服务类 GitHubService，本例使用 GitHub 的开放服务接口，获取 GitHub 的用户信息和用户

的仓库信息。主机地址 ENDPOINT 是 GitHub 的 API 地址，即 https://api.github.com；用于获取用户信息的请求 getUserData 是 Get 请求，子地址/users/{user}，使用@Path 设置 user 参数，返回 RxJava 的观察者对象，数据结构是 GitHubUser；用于获取用户的仓库信息的请求 getRepoData 也是 Get 请求，子地址/users/{user}/repos，同样使用@Path 设置 user 参数，返回 RxJava 的 Observable 的观察者对象，数据结构是 GitHubRepo 数组。

```
    public interface GitHubService {
        String ENDPOINT = "https://api.github.com";

        @GET("/users/{user}") // 获取用户信息
        Observable<UserListAdapter.GitHubUser>  getUserData(@Path("user")  String
user);

        @GET("/users/{user}/repos") // 获取用户的仓库信息，数组
        Observable<RepoListAdapter.GitHubRepo[]> getRepoData(@Path("user") String
user);
    }
```

用户的详情页面与用户列表页面的逻辑相似，由于篇幅原因，不一一展开说明，读者可以根据源码自行理解。

最终的 GitHub 用户列表页面，如图 4-6 所示。

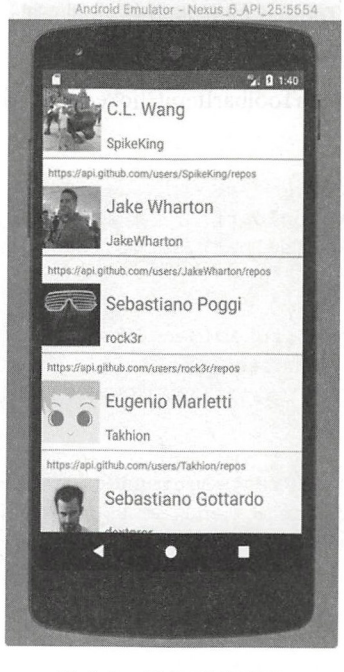


图 4-6　用户列表页面

4.1.6 控件的异步事件

在 Android 项目中，常见的异步任务除了网络请求，还有控件调用。每个控件都需要绑定一个方法，当用户触发控件的事件时，执行绑定的方法，给用户以反馈。RxJava 也为异步事件处理提供了相应的支持，由 RxBinding 库负责处理。RxJava 比传统异步事件的处理方式更加简洁，有些复杂的异步事件需要回调多个接口，如 EditText 的修改文字接口，包含修改前、修改完成、修改后三个接口。而在开发时，有时仅仅需要使用一个接口，如修改完成，但是在传统方式中，其他接口也需要声明一个空方法，显得代码非常冗余，而 RxJava 只需要声明真正需要的接口，其他的接口不需要声明。

RxBinding 也需要添加额外的依赖库。

```
compile 'com.jakewharton.rxbinding:rxbinding:0.3.0'
compile 'com.jakewharton.rxbinding:rxbinding-appcompat-v7:0.3.0'
compile 'com.jakewharton.rxbinding:rxbinding-design:0.3.0'
```

本例基于初始的 HelloWorld 页面，将其改造成 RxJava 的异步事件处理方法，主要含有 Toolbar 控件、Fab 控件、Snackbar 控件，同时为了对比 RxJava 与传统处理方式，使用两个 EditText 控件用于输入文本，一个 TextView 控件用于实时地显示被输入的文本。通过 ButterKnife 库，注入布局中控件的 ID，简化控件初始化的流程。

在 initToolbar()方法中，实现 Toolbar 的异步事件处理。通过 setSupportActionBar()将 Toolbar 加载至 Activity 页面，调用 RxToolbar 的 itemClicks()方法，绑定控件 mTToolbar，接着调用 subscribe()方法，响应控件的行为 onToolbarItemClicked，即当点击菜单的某一项时，输出这一项的标题（Title）。

```
private void initToolbar() {
    setSupportActionBar(mTToolbar); // 添加 Toolbar
    RxToolbar.itemClicks(mTToolbar).subscribe(this::onToolbarItemClicked);
}

private void onToolbarItemClicked(MenuItem menuItem) {
    String m = "点击\"" + menuItem.getTitle() + "\"";
    Toast.makeText(this, m, Toast.LENGTH_SHORT).show();
}

@Override public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.menu_rxbinding, menu);
    return super.onCreateOptionsMenu(menu);
}
```

在 initFabButton()方法中，实现 Fab 按钮的异步事件处理。调用 RxView 的 clicks()方法，绑定控件 mFabFab，接着调用 subscribe()方法，响应控件的行为 onFabClicked，即当点击 Fab 按钮

时，在 Snackbar 中给予提示。同样地，调用 RxSnackbar 的 dismisses()方法，绑定控件 snackbar，接着调用 subscribe()方法，响应控件的行为 onSnackbarDismissed，即当 Snackbar 消失时，给予提示。

```
    // 初始化 Fab 按钮
    private void initFabButton() {
        RxView.clicks(mFabFab).subscribe(this::onFabClicked);
    }

    // 点击 Fab 按钮
    private void onFabClicked(Void v) {
        Snackbar snackbar = Snackbar.make(findViewById(android.R.id.content), "点
击 Snackbar", Snackbar.LENGTH_SHORT);
        snackbar.show();
        RxSnackbar.dismisses(snackbar).subscribe(this::onSnackbarDismissed);
    }

    // 销毁 Snackbar, event 参考{Snackbar}
    private void onSnackbarDismissed(int event) {
        String text = "Snackbar 消失代码:" + event;
        Toast.makeText(this, text, Toast.LENGTH_SHORT).show();
    }
```

最后，在 initEditText()方法中，监听 EditText 的内容改变，对比传统的异步处理方式与 RxJava 的异步处理方式的区别。传统的异步处理方式声明匿名类 TextWatcher，实现三个接口 beforeTextChanged()、onTextChanged()、afterTextChanged()，由于示例仅仅需要监听文字的变化，在 onTextChanged()中设置 TextView 的显示，其余 beforeTextChanged()和 afterTextChanged()都是空方法；而 RxJava 的异步处理方式调用 RxTextView 的 textChanges()方法，设置被监听的 EditText，接着调用 subscribe()方法设置 TextView 的显示，不需要声明无用的接口，此处的 textChanges()方法就相当于匿名类的 onTextChanged()方法。由此，可以感受到 RxJava 方式的流畅与简洁。

```
    private void initEditText() {
        // 正常方式
        mEtUsualApproach.addTextChangedListener(new TextWatcher() {
            @Override
            public void beforeTextChanged(CharSequence s, int start, int count, int
after) {
            }

            @Override public void onTextChanged(CharSequence s, int start, int before,
int count) {
                mTvShow.setText(s);
            }

            @Override public void afterTextChanged(Editable s) {
            }
```

```
        });

        // Rx 方式
        RxTextView.textChanges(mEtReactiveApproach).subscribe(mTvShow::setText);
    }
```

控件的异步事件的处理页面如图 4-7 所示。

图 4-7　控件的异步事件的处理页面

4.1.7　线程安全

在 Android 的应用中，当使用异步线程时，需要防范内存泄漏。最常见的内存泄漏例子是某个页面启动一个线程，处理时间超长或者无限循环的任务，当页面关闭时，线程仍在执行任务，无法释放，最终导致内存泄漏。其解决方案是当页面关闭时，由此页面创建的异步线程也全部关闭。

RxJava 作为处理异步任务的工具，自然也需要考虑内存泄漏的问题。RxJava 提供一个开源库 RxLifecycle，将异步任务的线程绑定至页面的生命周期中，当页面关闭时，将全部的异步线程关闭，避免内存泄漏。RxLifecycle 需要依赖相应的 maven 库，如：

```
compile 'com.trello:rxlifecycle:0.4.0'
compile 'com.trello:rxlifecycle-components:0.4.0'
```

为了模拟内存泄漏的情况，本例创建一个无限循环的计时器，时间每隔一秒显示一次，如果未进行内存泄漏的防范，当关闭页面时，计时器仍然会无限地执行；如果将线程绑定至生命周期，当关闭页面时，则计时器也同时关闭。RxJava 需要绑定 Activity 的生命周期，RxAppCompatActivity 替换 AppCompatActivity 成为 Activity 的继承类，同时，在 RxJava 的信息流中，添加 compose，调用 RxAppCompatActivity 父类的 bindToLifecycle()方法，完成 Observable 与 Activity 生命周期的绑定。当页面关闭时，即执行 onPause()方法，Observable 会自动结束当前线程的循环任务，避免发生内存泄漏。

```
    public class SafeActivity extends RxAppCompatActivity {
        private static final String TAG = "DEBUG-WCL: " +
SafeActivity.class.getSimpleName();

        @Bind(R.id.simple_tv_text) TextView mTvText;

        @Override protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_simple);
            ButterKnife.bind(this);

            Observable.interval(1, TimeUnit.SECONDS)
                    .observeOn(AndroidSchedulers.mainThread())
                    .compose(bindToLifecycle()) // 管理生命周期，防止内存泄露
                    .subscribe(this::showTime);
        }

        private void showTime(Long time) {
            mTvText.setText(String.valueOf("时间计数: " + time));
            Log.d(TAG, "时间计数器: " + time);
        }

        @Override
        protected void onPause() {
            super.onPause();
            Log.w(TAG, "页面关闭!");
        }
    }
```

定时器的页面如图 4-8 所示。

图 4-8　定时器的页面

在 Android 开发中，RxJava 是处理异步任务最好的方式，有些 RxJava 的重度用户甚至说，“RxJava is everything!”(RxJava 就是全部，表示 RxJava 可以开发所有功能)。在程序开发过程中，异步任务总是让人们头疼，任务开始之后，不限时的等待反馈（异步），还需要处理若干异常的状态，而 RxJava 的出现，正是解决这一问题的核心。

（1）RxJava 从观察者模式中受到启发，创建出一种基于流的异步处理方式，每一种常见的改变都有相应的范式，通过标准范式与具体方法的结合，形成不同的功能，像流水一样，依次进行。

（2）每一个流的加工函数，都需要一个匿名类实现具体的功能，这样就会导致匿名类过多，代码臃肿。可以使用 Lambda 表达式开发 RxJava，将匿名类简化为方法引用，使得数据流更加简洁。

（3）异步任务最常见的两种形式就是网络请求与事件响应，RxJava 与 Retrofit 结合，创建出基于流的异步网络请求的处理方式；RxJava 与 RxBinding 结合，创建出基于流的异步事件响应的处理方式；使得异步开发过程更加优雅和流畅。

（4）只要涉及异步事件，就不可避免地需要注意线程安全，异步事件一般都是基于单独的监听线程，当主页面（主线程）关闭时，长时间或循环的异步事件可能仍在执行，就会导致内存泄漏。RxJava 与 RxLifecycle 结合，在关闭主页面时，同时停止异步事件的监听线程，保证线程安全。

RxJava 的方法纷繁复杂，辅助库层出不穷，但是都属于本文的这几个类别，对于开源库的理解，首先要理解整个框架的运行原理，其次要理解不同功能的几个类别，最后才是深研各个开发细节。本文对于 RxJava 的介绍跳出了 RxJava 的实现与细节，分析整个框架的运行原理与功能类别，避免所谓的“不识庐山之面目”。希望喜欢 RxJava 的读者能够继续深研下去，这一定会是一段充满乐趣的学习之旅。

4.2　全面解析依赖注入库 Dagger 的使用方式

Dagger 在编译时执行依赖注入（Dependency Injection，DI）的静态框架，最初由 Square 公司维护，进而交由 Google 维护，即加入 Google I/O。Dagger 的含义是有向非循环图，即 Directed Acyclic Graph，DAGger。依赖注入最初基于反射（Reflection）实现，而 Dagger 主要解决由反射导致的开发和性能上的问题。

依赖注入是指在开发项目中，经常需要在一个对象 A 中创建另一个对象 B 的实例，这种行为会导致 A 依赖于 B，也是产生耦合的常见形式。对于一个大型项目而言，过多的依赖会导致代码难以维护，经常会碰到修改一个小需求，而需要大面积地修改代码的情况，使得代码难以维护。因此，如果在类中不通过创建实例的方式依赖对象，而是通过提供的参数注入方法，将需要依赖的类和提供依赖的类，实现隔离，降低耦合。反射也可以实现依赖注入功能，只不过注入时机是在运行期间，而 Dagger 的注入时机是在编译期间。Dagger 的本质是一个依赖注入的框架，自动生成代码，负责创建依赖对象。

Dagger 的优点：

◎　访问全局对象实例：提供访问全局对象引用的简易方式，被声明为单例的实例均可使用注解进行访问。

◎　配置复杂依赖关系：通过依赖关系自动生成易于分析的代码，不需关注类的实例创建顺序，传递实例引用至相应的方法参数。

◎　测试单元模块便捷：将类的依赖关系独立出来，可以抽取不同的模块进行测试，将依赖的注入和配置独立于组件之外。

本文基于Dagger的2.x版本，讲解Dagger的知识要点和编程细节，Dagger的源码参考Google的 GitHub 开源项目：https://google.github.io/dagger/，如图 4-9 所示。

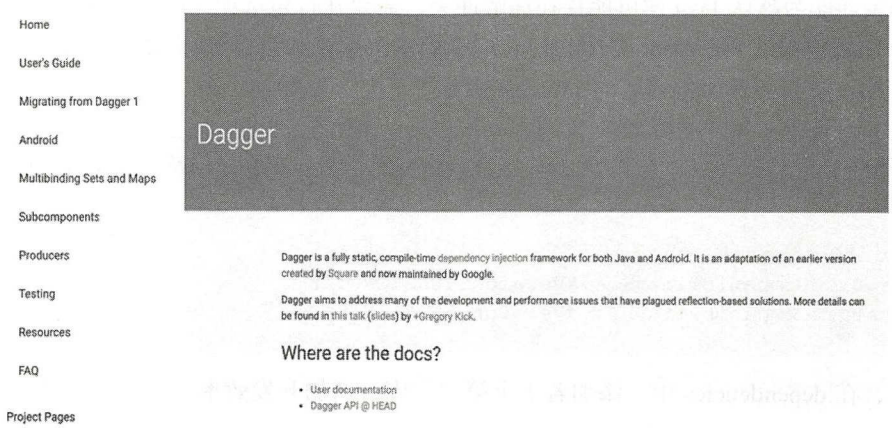


图 4-9　Dagger 源码

本文实例完整代码的下载地址：https://github.com/SpikeKing/wcl-dagger-demo。

4.2.1 工程配置

本例基于 Android Studio 中一个简单的 HelloWorld 项目。在工程的 build.gradle 中，添加所需的依赖库。第一个是构建脚本（Build Script）android-apt，用于静态编译 Android 项目中的类，负责 Dagger 的在编译期间注入依赖。

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'
    }
}
```

第二个是插件 retrolambda，用于在 Java 的低版本中，支持 Lambda 表达式。在开发过程中，使用 Lambda 表达式，可以减少匿名类的编写，使代码更简洁。

```
plugins {
    id "me.tatarka.retrolambda" version "3.2.4"
}
```

接着，为工程加载默认应用插件 application 和静态编译插件 android-apt，静态编译插件依赖于引入的构建脚本。

```
apply plugin: 'com.android.application'
apply plugin: 'com.neenbedankt.android-apt'
```

静态编译库与默认 Java 库可能存在注解冲突，需要在打包选项（Packaging Options）中，排除 Java 注解包中的 Processor 类。增加 Java 1.8 版本的编译和目标源，需要在编译选项（Compile Options）中，支持源和目标兼容的 Java 1.8 版本。

```
packagingOptions {
    exclude 'META-INF/services/javax.annotation.processing.Processor'
}

compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
}
```

最后，在 dependencies 中，添加若干个第三方库，增加开发效率，避免重复造轮子。

◎ Appcompat-v7：提供默认的 Android API 兼容库。

◎ RecyclerView：提供 RecyclerView 控件，一种耦合较弱的列表控件。
◎ Butterknife：提供布局 XML 文件至逻辑 Java 文件的 ID 映射的支持。
◎ Dagger2：提供依赖注入的支持。
◎ Rx 系列：提供 RxAndroid 和 RxJava 的组件，一种响应式开发框架。
◎ Retrofit 系列：提供基于 RxJava 的适配器、基于 Gson 的转换器的 REST 网络请求的支持。
◎ Annotation：提供 Java 注释解析的支持。

```
    dependencies {
        compile fileTree(dir: 'libs', include: ['*.jar'])
        testCompile 'junit:junit:4.12'
        compile 'com.android.support:appcompat-v7:23.1.1'
        compile 'com.android.support:recyclerview-v7:23.1.1' // RecyclerView
        compile 'com.jakewharton:butterknife:7.0.1' // 标注
        compile 'com.google.dagger:dagger:2.0.2' // Dagger2
        compile 'com.google.dagger:dagger-compiler:2.0.2' // Dagger2
        compile 'io.reactivex:rxandroid:1.1.0' // RxAndroid
        compile 'io.reactivex:rxjava:1.1.0' // 推荐同时加载 RxJava
        compile 'com.squareup.retrofit:retrofit:2.0.0-beta2' // Retrofit 网络处理
        compile 'com.squareup.retrofit:adapter-rxjava:2.0.0-beta2' // Retrofit 的
RX 解析库
        compile 'com.squareup.retrofit:converter-gson:2.0.0-beta2' // Retrofit 的
Gson 库
        provided 'javax.annotation:jsr250-api:1.0' // Java 标注
    }
```

目前，Android 工程是基于 Gradle 框架进行架构，工程配置均设置在 build.gradle 中。

4.2.2 主页逻辑

主页 MainActivity.java 的逻辑非常简单，setContentView()设置主页布局，ButterKnife 绑定布局 ID，DemoApplication 注入依赖组件，提供按钮的逻辑接口 gotoReposList()，用于跳转项目列表的详情页。其中，依赖注入的组件 DemoApplication，无实际功能，仅提供开发示例。

```
    public class MainActivity extends AppCompatActivity {
        @Override
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main);

            ButterKnife.bind(this);

            DemoApplication.component().inject(this); // 应用注入
        }
```

```
    // 跳转列表视图
    public void gotoReposList(View view) {
        startActivity(new Intent(this, ReposListActivity.class));
    }
}
```

Dagger 主要含有三个部分，即注入（Inject）、模块（Module）、组件（Component）：

◎ 注入负责在类中注入依赖，将类的实例注入至声明。

◎ 模块负责提供支持依赖注入类的实例。

◎ 组件负责将模块注册入类中。

首先创建组件接口的父类 DemoGraph，类中含有注入的接口方法 inject，参数是需要被注入依赖的类。

```
public interface DemoGraph {
    void inject(MainActivity mainActivity); // 注入 MainActivity
    void inject(ReposListActivity reposListActivity); // 注入列表 Activity
}
```

继承依赖注入接口的类是组件类 DemoComponent，也是接口。DemoComponent 接口通过@Component 注解绑定模块类 MainModule.class 和 ApiModule.class，通过@Singleton 注解设置为单例模式。接着构建项目，自动生成模板类 DaggerDemoComponent，这样就可以在编译期间执行依赖注入。在 DemoComponent 接口中，添加初始化类 Initializer，用于初始化模块的参数，设置构造方法为私有，提供静态接口 init()，将所需参数输入模块，完成模块中类的实例化。例如，MainModule 模块需要 Application 参数初始化，则在 init()接口中传递参数，调用 DaggerDemoComponent#builder#mainModule 方法，传入 MainModule 的实例，最后调用 build 方法，完成组件的创建；而 ApiModule 模块不需要额外参数初始化，则直接在构造器中创建。

```
@Singleton
@Component(modules = {MainModule.class, ApiModule.class})
public interface DemoComponent extends DemoGraph {
    final class Initializer {
        private Initializer() {
        } // No instances.

        // 初始化组件
        public static DemoComponent init(DemoApplication app) {
            return DaggerDemoComponent.builder()
                    .mainModule(new MainModule(app))
                    .build();
        }
    }
}
```

重写项目的 Application 类，在入口函数 onCreate()中调用 buildComponentAndInject()，实例化组件的接口类 sDemoGraph，同时提供访问接口 component()，用于绑定注入依赖的类，即 DemoGraph 的 inject()方法。

```
public class DemoApplication extends Application {
    private static DemoGraph sDemoGraph;
    private static DemoApplication sInstance;

    @Override public void onCreate() {
        super.onCreate();
        sInstance = this;
        buildComponentAndInject();
    }

    public static DemoGraph component() {
        return sDemoGraph;
    }

    public static void buildComponentAndInject() {
        sDemoGraph = DemoComponent.Initializer.init(sInstance);
    }
}
```

主模块 MainModule 使用@Module 注解标记，提供单例 Application 和单例 Resources 的访问接口。模块对外提供实例的接口使用@Provides 注解标记，接口名使用 provide 前缀，静态编译的模板创建类根据前缀识别接口，创建实例，用于依赖注入。单例接口的一般命名规则是【provide+单例类名】，如 provideApplication 提供 Application 的单例实例。单例模式使用@Singleton 注解标记。

```
@Module
public class MainModule {
    private final DemoApplication mApp;

    public MainModule(DemoApplication application) {
        mApp = application;
    }

    @Provides @Singleton
    protected Application provideApplication() {
        return mApp;
    }

    @Provides @Singleton
    protected Resources provideResources() {
        return mApp.getResources();
    }
}
```

API 模块 ApiModule 与主模块 MainModule 类似，同样使用@Module 注解标记，提供网络访问服务类 GitHubService 的实例。

主页的样式非常简单，仅含有一个跳转按钮。

图 4-10　MainActivity

4.2.3　详情逻辑

详情逻辑页 ReposListActivity 基于 RecyclerView 的列表样式，展示 GitHub 用户的项目详情列表（Repositories）。使用 ButterKnife 的@Bind 注解绑定列表视图 RecyclerView；将 ReposListActivity 绑定至依赖注入组件 DemoApplication；设置列表 RecyclerView 为竖直样式，并且添加适配器 ListAdapter。

ReposListActivity 类提供加载网络数据的方法 loadData()，使用响应式编程 Retrofit+RxJava 框架，调用 GitHubService#getRepoData()方法获取订阅源，调用 subscribeOn()设置订阅线程为新线程（newThread），调用 observeOn()设置观察源为主线程（mainThread），最后将已订阅完成的数据输入至 ListAdapter#setRepos()方法中。

需要注意的是：

◎ GitHubService 实例通过依赖注入初始化，使用@Inject 注解自动注入，而非实例化（New）方法。

◎ adapter::setRepos 是 Lambda 表达式的方法引用（Method References）样式，当方法的参数和返回值与匿名类函数相同时，即可使用。

```
public class ReposListActivity extends Activity {
    @Bind(R.id.repos_rv_list) RecyclerView mRvList;
    @Inject GitHubService mGitHubService;

    @Override protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```

```
        setContentView(R.layout.activity_repos_list);
        ButterKnife.bind(this);

        DemoApplication.component().inject(this);

        LinearLayoutManager manager = new LinearLayoutManager(this);
        manager.setOrientation(LinearLayoutManager.VERTICAL);
        mRvList.setLayoutManager(manager);

        ListAdapter adapter = new ListAdapter();
        mRvList.setAdapter(adapter);
        loadData(adapter);
    }

    // 加载数据
    private void loadData(ListAdapter adapter) {
        mGitHubService.getRepoData("SpikeKing")
                .subscribeOn(Schedulers.newThread())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(adapter::setRepos);
    }
}
```

网络服务 GitHubService 接口的样式是 Retrofit+RxJava 框架，接口提供网址 ENDPOINT，提供网络访问的 getRepoData()接口，GitHubService 接口的实现位于其实例化方法中。

◎ 接口的声明使用注解@GET("/users/{user}/repos")表示含有参数 user 的子地址。

◎ 接口的参数使用注解@Path("user")，表示参数的 user 与子地址的 user 相对应，类似于 String Format 样式。

◎ 接口的返回值，使用 RxJava 的观察者样式，即返回可被订阅的观察者。当异步网络访问完成时，观察者会向订阅者发送 ListAdapter.Repo 的列表数据。

```
public interface GitHubService {
    String ENDPOINT = "https://api.github.com";

    @GET("/users/{user}/repos")
    Observable<ArrayList<ListAdapter.Repo>> getRepoData(@Path("user") String
user);
}
```

本例是依赖注入架构，GitHubService 接口实例化位于 ApiModule 类的 provideGitHubService()方法中。Builder()创建 Retrofit 的构造器，baseUrl()设置网络地址 URL，addCallAdapterFactory()添加 RxJava 适配器，addConverterFactory()添加 Gson 数据转换器，最终创建配置完成的 Retrofit 类。通过 Retrofit 的 create()函数和 GitHubService.class 接口类，创建 GitHubService 实例。最终，通过 Dagger 的 ApiModule#provideGitHubService()接口，将 GitHubService 实例注入至所需的类

中。

```
@Module
public class ApiModule {
    @Provides @Singleton
    protected GitHubService provideGitHubService() {
        Retrofit retrofit = new Retrofit.Builder()
                .baseUrl(GitHubService.ENDPOINT)
                .addCallAdapterFactory(RxJavaCallAdapterFactory.create()) // 添加 Rx 适配器
                .addConverterFactory(GsonConverterFactory.create()) // 添加 Gson 转换器
                .build();
        return retrofit.create(GitHubService.class);
    }
}
```

RecyclerView 适配器类 ListAdapter，继承 RecyclerView.Adapter 基类，模板参数 ListAdapter.RepoViewHolder。

```
public class ListAdapter extends RecyclerView.Adapter<ListAdapter.RepoViewHolder>
```

在适配器类 ListAdapter 中，保存项目信息列表 ArrayList，提供设置列表的接口 setRepos()。接着，实现适配器基类的三个接口：onCreateViewHolder()用于创建 ViewHolder，onBindViewHolder()用于将 ViewHolder 绑定数据，getItemCount()用于获取列表行数。

```
private ArrayList<Repo> mRepos; // 项目信息
public ListAdapter() {
    mRepos = new ArrayList<>();
}

public void setRepos(ArrayList<Repo> repos) {
    mRepos = repos;
    notifyItemInserted(mRepos.size() - 1);
}

@Override
public RepoViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
    View view = LayoutInflater.from(parent.getContext())
            .inflate(R.layout.item_repo, parent, false);
    return new RepoViewHolder(view);
}

@Override public void onBindViewHolder(RepoViewHolder holder, int position) {
    holder.bindTo(mRepos.get(position));
}

@Override public int getItemCount() {
```

```
        return mRepos.size();
    }
```

在适配器类 ListAdapter 的 RepoViewHolder 中，使用 ButterKnife 将布局绑定至类中，在 bindTo()方法中，设置列表项的标题 mIvRepoName 和内容 mIvRepoDetail。

```
    public static class RepoViewHolder extends RecyclerView.ViewHolder {
        @Bind(R.id.item_iv_repo_name) TextView mIvRepoName;
        @Bind(R.id.item_iv_repo_detail) TextView mIvRepoDetail;

        public RepoViewHolder(View itemView) {
            super(itemView);
            ButterKnife.bind(this, itemView);
        }

        public void bindTo(Repo repo) {
            mIvRepoName.setText(repo.name);
            mIvRepoDetail.setText(String.valueOf(repo.description + "(" +
repo.language + ")"));
        }
    }
```

项目 Repo 类负责存储项目信息，在 GitHub 服务中，获取项目的项目名称、描述、语言，通过 RPC 的 Gson 转换器，直接映射至 Repo 类的同名字段。

```
    public static class Repo {
        public String name; // 项目名称
        public String description; // 描述
        public String language; // 语言
    }
```

最终，本例获取 GitHub 中"SpikeKing"用户的项目列表信息，在页面中展示，如图 4-11 所示。

Dagger 的核心是解决实例与实例之间的依赖，通过依赖注入的设计模式，降低类之间的耦合性，提高项目的可扩展性和可维护性。当一个对象需要或依靠其他对象协调工作时，就会产生依赖关系。如果将创建依赖对象的实例过程，位于不同模块中，当依赖对象被修改时，不必修改已有被依赖对象的代码，提高项目的可维护性。Dagger 的核心是三个部分：组件、模块、注入。

- ◎ 模块负责依赖类的实例化，提供访问接口。
- ◎ 组件负责创建模块（Module），连接依赖类与被依赖类。
- ◎ 注入负责将模块的实例化类赋值给被依赖类的引用。

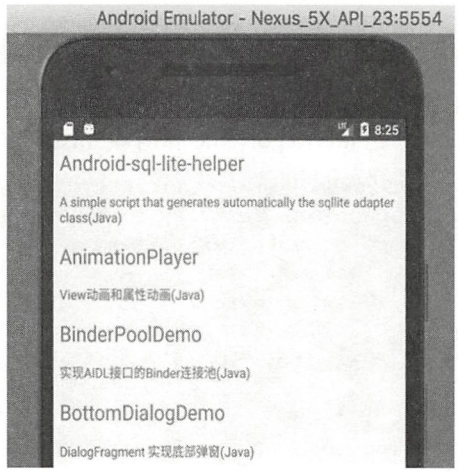


图 4-11　详情页面

同时，在 Dagger 开发框架中，使用注解（即“@”）简化开发逻辑，通过静态编译脚本，在编译时期即可完成依赖注入，减少在运行时期中通过反射的时间损耗。

本文的实例仅仅提供一种开发范式，在讲解 Dagger 原理的同时，引入 Retrofit 和 RxJava，将开发框架的各个模块通过实例串联起来。Dagger 就像一把锋利的“匕首”，提高开发的效率和并确保稳定，是构建大型 Android 项目必不可少的代码库之一。“君子生非异也，善假于物也。”多利用工具，能早日成为合格的高级程序员。

4.3　基于响应式编程的网络数据同步及缓存框架

响应式编程的核心功能就是在于处理异步任务，其中在网络访问中，几乎全部请求都是异步请求。因此，响应式编程非常适合处理异步的网络数据。网络访问是非常消耗流量和电量的过程，因此缓存是网络框架中必备的一项功能。当用户的网络连接异常时，也可以给用户返回以前加载过的数据，避免空白页面，提升用户体验。

比较常见的缓存模式有：

◎　第一次加载：当网络通畅时，等待下载，请求网络数据，存储至数据库，再展示已存数据；当网络异常时，放弃请求网络数据。

◎　其他次加载：当网络通畅时，读取数据库，展示已存数据，同时请求网络数据，存储至数据库，再同步展示已存数据；当网络异常时，读取数据库，展示已存数据，放弃请求网络数据。

展示的数据均来源于数据库，保证数据的一致性，当网络访问异常时，仍有展示的数据，

避免空数据的页面。这样，就是一个基础的网络数据同步及缓存框架。

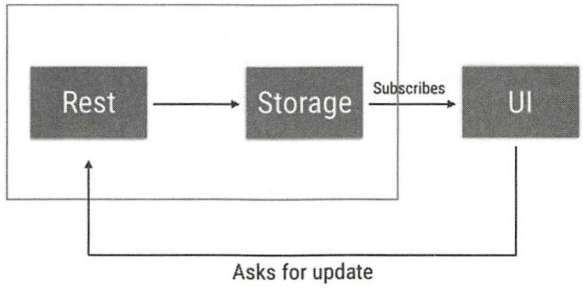


图 4-12　网络数据同步及缓存框架

响应式编程的核心组件是 Dagger+Retrofit+RxJava 的标准组合。本节通过实例来分析如何通过响应式编程的方式实现网络数据同步及缓存框架。

本文实例完整代码的下载地址为 https://github.com/SpikeKing/wcl-rx-cache-demo。

4.3.1　工程配置

本例是在模拟器的环境中执行，数据源来自开放的 GitHub 标准 API，因此需要模拟器访问网络。如果模拟器无法访问网络，可能是模拟器的 DNS 与实际的 DNS 不符，需要重新配置。配置的方式有很多种，常用的方式是在启动时增加“-dns-server”参数，例如：

```
emulator -avd Nexus_5X_API_27 -dns-server 8.8.8.8
```

即，启动名为 Nexus_5X_API_27 的虚拟机，DNS 服务器为“8.8.8.8”。emulator 命令位于“android-sdk/tools/”文件夹下，这样虚拟机就支持访问网络了。

本例依旧是从一个简易的 Hello World 工程开始，首先修改工程配置 build.gradle，添加响应式编程的核心组件是 Dagger+Retrofit+RxJava 的依赖库，和一些常用的第三方开源库，用于加快开发速度，优化代码逻辑。

依赖注入的静态编译插件 android-apt：

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'
    }
}
```

```
apply plugin: 'com.neenbedankt.android-apt'
```

支持 Lambda 表达式的插件 retrolambda：

```
plugins {
    id 'me.tatarka.retrolambda' version '3.2.4'
}

android {
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
}
```

布局 ID 注入的库 butterknife：

```
compile 'com.jakewharton:butterknife:7.0.1' // ButterKnife
```

接着，添加 Dagger+Retrofit+RxJava 组合的依赖库。

```
// RxAndroid
compile 'io.reactivex:rxandroid:1.1.0'

// Dagger
compile 'com.google.dagger:dagger:2.0.2'
compile 'com.google.dagger:dagger-compiler:2.0.2'

// Retrofit
compile 'com.squareup.retrofit:retrofit:2.0.0-beta2'
compile 'com.squareup.retrofit:converter-gson:2.0.0-beta2'
compile 'com.squareup.retrofit:adapter-rxjava:2.0.0-beta2'
compile 'com.squareup.okhttp:okhttp:2.4.0'
```

至此，工程配置已经添加完成。

4.3.2 首页

在首页布局 activity_main 中，含有两个跳转按钮，非常简洁：

◎ main_b_nocache 是非缓存的网络请求页面的跳转按钮。

◎ main_b_cache 是有缓存的跳转按钮。

```
<LinearLayout
    android:orientation="vertical"
    tools:context="org.wangchenlong.wcl_rx_cache_demo.MainActivity">

    <Button
```

```
        android:id="@+id/main_b_nocache"
        android:onClick="gotoNoCache"
        android:text="@string/use_nocache"/>

    <Button
        android:id="@+id/main_b_cache"
        android:onClick="gotoCache"
        android:text="@string/use_cache"/>
</LinearLayout>
```

在首页逻辑 MainActivity 中，包含非缓存页面的逻辑 gotoNoCache()和跳转缓存页面的逻辑 gotoCache()。其中，缓存页面是 CacheActivity，非缓存页面是 NocacheActivity。

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    // 跳转无缓存
    public void gotoNoCache(View view) {
        startActivity(new Intent(this, NocacheActivity.class));
    }

    // 跳转有缓存
    public void gotoCache(View view) {
        startActivity(new Intent(this, CacheActivity.class));
    }
}
```

首页的效果如图 4-13 所示。

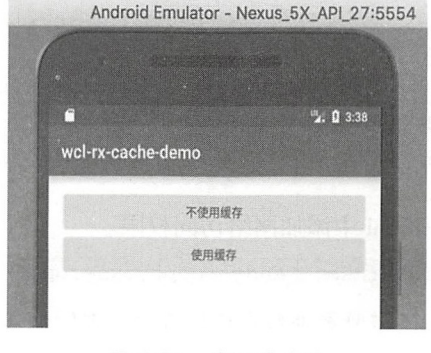


图 4-13　首页的效果

4.3.3 数据源

无论是有缓存页面，还是无缓存页面，都需要基础的网络请求功能。首先，创建 Retrofit 的网络服务接口类 GitHubService，Host 地址是 https://api.github.com，访问路径是 /users/{user}/repos，参数是用户名 user，返回的数据结构是 RxJava 的观察者 Observable<ArrayList<Repo>>。即从 GitHub 的标准 API 中，获取 GitHub 用户仓库列表的信息。

```
public interface GitHubService {
    String BASE_URL = "https://api.github.com";
    @GET("/users/{user}/repos")  Observable<ArrayList<Repo>>  getRepos(@Path
("user") String user);
}
```

例如，SpikeKing 的仓库信息：

```
https://api.github.com/users/SpikeKing/repos。
```

接着，基于网络服务接口类 GitHubService 创建客户端 GitHubClient，GitHubClient 是非常标准的客户端模式。Retrofit 创建 GitHubService 接口的实例 mGitHubService，调用 baseUrl()设置基础 URL，调用 addConverterFactory()添加 Gson 网络数据转换器，调用 addCallAdapterFactory()添加 RxJava 网络数据适配器。提供对外访问接口 getRepos()，获取 GitHub 用户的仓库列表信息。

```
public class GitHubClient {
    private GitHubService mGitHubService;

    public GitHubClient() {
        mGitHubService = new Retrofit.Builder()
                .baseUrl(GitHubService.BASE_URL)
                .addConverterFactory(GsonConverterFactory.create())
                .addCallAdapterFactory(RxJavaCallAdapterFactory.create())
                .build()
                .create(GitHubService.class);
    }

    public Observable<ArrayList<Repo>> getRepos(String username) {
        return mGitHubService.getRepos(username);
    }
}
```

最后，在 AndroidManifest.xml 中添加网络访问权限。

```
<uses-permission android:name="android.permission.INTERNET"/>
```

除此之外，有的缓存页面还需要本地数据库的支持。本例创建一个基于 RxJava 的数据库的异步访问源 ObservableRepoDb。在数据源 ObservableRepoDb 中，创建仓库列表的 RxJava 主题 mPublishSubject，创建 RepoDbHelper 数据管理器 mDbHelper。

```
public class ObservableRepoDb {
    private PublishSubject<ArrayList<Repo>> mPublishSubject; // 发表主题
    private RepoDbHelper mDbHelper; // 数据库

    public ObservableRepoDb(Context context) {
        mDbHelper = new RepoDbHelper(context);
        mPublishSubject = PublishSubject.create();
    }

    // ...
}
```

推荐使用 Jinja2 脚本自动生成数据库处理类 RepoDbHelper，其使用方式可以参考：

```
Jinja2: https://github.com/fedepaol/Android-sql-lite-helper
```

数据源 ObservableRepoDb 提供数据库的异步访问接口 getObservable()，将数据库中仓库信息封装成 RxJava 的异步观察者 Observable，调用 concatWith()将观察者与主题变量 mPublishSubject 关联起来。主题 mPublishSubject 的主要作用是当向数据库中插入数据时，同时，更新观察者的数据，保证数据库与展示数据的一致性。在私有方法 getRepoList()中，负责扫描数据库，获取仓库列表数据。

在其他情况中，当数据量较大时，访问数据库的时间就会很长，如果直接获取数据，则会造成页面的等待，容易产生 ANR 情况，页面被系统提示无响应，应用可能被用户直接杀死。但是，如果修改为异步获取数据，则不会造成等待，继续执行其他逻辑，当数据获取完成后，再异步地修改页面。这就是异步处理的优点，而 RxJava 的出现，极大地改善了异步逻辑的开发效率。

```
public Observable<ArrayList<Repo>> getObservable() {
    Observable<ArrayList<Repo>>  firstObservable  =  Observable.fromCallable
(this::getRepoList);
    return firstObservable.concatWith(mPublishSubject); // 连接发表主题
}

private ArrayList<Repo> getRepoList() {
    mDbHelper.openForRead();
    ArrayList<Repo> repos = new ArrayList<>();
    Cursor c = mDbHelper.getAllRepo();
    if (!c.moveToFirst()) {
        return repos; // 返回空
    }

    do {
        // 添加数据
        repos.add(new Repo(
                c.getString(RepoDbHelper.REPO_ID_COLUMN_POSITION),
```

```
                c.getString(RepoDbHelper.REPO_NAME_COLUMN_POSITION),
                c.getString(RepoDbHelper.REPO_DESCRIPTION_COLUMN_POSITION),
                new
Repo.Owner(c.getString(RepoDbHelper.REPO_OWNER_COLUMN_POSITION), "", "", "")));
        } while (c.moveToNext());
        c.close();
        mDbHelper.close();
        return repos;
    }
```

除了访问数据，数据源 ObservableRepoDb 还提供写入数据的接口 insertRepoList()，将仓库列表信息 repos 逐条地写入数据库 mDbHelper。当写入完成后，调用主题 mPublishSubject 的 onNext()方法，将数据发送到异步的访问接口中。

```
    public void insertRepoList(ArrayList<Repo> repos) {
        mDbHelper.open();
        mDbHelper.removeAllRepo();
        for (Repo repo : repos) {
            mDbHelper.addRepo(
                    repo.getId(),
                    repo.getName(),
                    repo.getDescription(),
                    repo.getOwner().getLogin()
            );
        }
        mDbHelper.close();
        mPublishSubject.onNext(repos); // 会调用更新数据
    }
```

这样就保证写入数据库中的数据与展示的数据保持一致，在写入的同时调用读取，也是 RxJava 的一个比较实用的异步逻辑。

```
    firstObservable.concatWith(mPublishSubject);
    mPublishSubject.onNext(repos);
```

4.3.4 依赖注入

网络数据源与数据库数据源已经准备完成。使用 Dagger 的依赖注入方式，将两个数据源注入到类中。首先，创建 Module 模块 ApiModule，提供 Application、GitHubClient（网络数据源）、ObservableRepoDb（数据库源）等三个注入类。

```
    @Module public class ApiModule {
        private Application mApplication;

        public ApiModule(Application application) {
            mApplication = application;
```

```
    }

    @Provides @Singleton
    public Application provideApplication() {
        return mApplication;
    }

    @Provides @Singleton
    GitHubClient provideGitHubClient() {
        return new GitHubClient();
    }

    @Provides ObservableRepoDb provideObservableRepoDb() {
        return new ObservableRepoDb(mApplication);
    }
}
```

接着，将 ApiModule 模块封装成 ApiComponent 组件，提供两个待注入依赖的类接口，即无缓存类 NocacheActivity，有缓存类 CacheActivity。同时，重新构建工程，根据 Dagger 模块自动生成静态的 DaggerApiComponent 类，用于依赖注入。

```
@Singleton @Component(modules = ApiModule.class)
public interface ApiComponent {
    void inject(NocacheActivity activity);

    void inject(CacheActivity activity);
}
```

最后，继承 Application 类，覆写 onCreate()接口，创建 ApiComponent 类，并提供访问接口 getApiComponent()。

```
public class RcApplication extends Application {
    private ApiComponent mApiComponent;

    @Override public void onCreate() {
        super.onCreate();
        mApiComponent = DaggerApiComponent.builder()
                .apiModule(new ApiModule(this)).build();
    }

    public ApiComponent getApiComponent() {
        return mApiComponent;
    }
}
```

修改 AndroidManifest.xml，重新 application 的 name 属性，使用 RcApplication 类。

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="org.wangchenlong.wcl_rx_cache_demo">
```

```
        <application
            android:name=".RcApplication">
        </application>
    </manifest>
```

4.3.5 无缓存模式

无缓存页面 NocacheActivity 主要用于和有缓存页面进行对比。当页面使用网络请求加载数据时，如果没有缓存，在每次加载数据时，遇到网络较长、加载时间较长，则会导致页面长时间处于加载状态，用户体验极差。

在页面 NocacheActivity 中，使用 RecyclerView 展示列表信息，获取 ApiComponent 组件绑定 NocacheActivity 页面，并向其中注入 Application 实例和 GitHubClient 实例。

```
    public class NocacheActivity extends Activity {
        @Bind(R.id.nocache_rv_list) RecyclerView mRvList;
        @Bind(R.id.nocache_pb_progress) ProgressBar mPbProgress;

        @Inject Application mApplication;
        @Inject GitHubClient mGitHubClient;

        private ListAdapter mListAdapter;

        @Override protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_nocache);
            ButterKnife.bind(this);

            ((RcApplication) getApplication()).getApiComponent().inject(this);

            LinearLayoutManager layoutManager = new LinearLayoutManager
(mApplication);
            mRvList.setLayoutManager(layoutManager);

            mListAdapter = new ListAdapter();
            mRvList.setAdapter(mListAdapter);
        }
        // ...
    }
```

当页面准备显示的时候，即 onResume()，获取 GitHub 用户 SpikeKing 的仓库列表信息，延迟 3 秒返回数据，用于模拟网络较差的效果，同时显示加载进程条。当请求数据成功时，将数据在列表中显示，同时隐藏加载进程条；当请求数据失败时，直接隐藏加载进程条。也就是说，当网络情况较差时，如本例中的延迟 3 秒，则用户将会经历“漫长”的 3 秒等待，一直只有观

看加载进程条的旋转，极大地影响用户体验。

```
@Override protected void onResume() {
    super.onResume();
    mGitHubClient.getRepos("SpikeKing")  // 延迟 3 秒，模拟网络较差的效果
            .delay(3, TimeUnit.SECONDS)
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(this::onSuccess, this::onError);

    mPbProgress.setVisibility(View.VISIBLE);
}

private void onSuccess(ArrayList<Repo> repos) {
    mListAdapter.setRepos(repos);
    mPbProgress.setVisibility(View.INVISIBLE);
}

private void onError(Throwable throwable) {
    mPbProgress.setVisibility(View.INVISIBLE);
}
```

加载进度条的效果如图 4-14 所示。

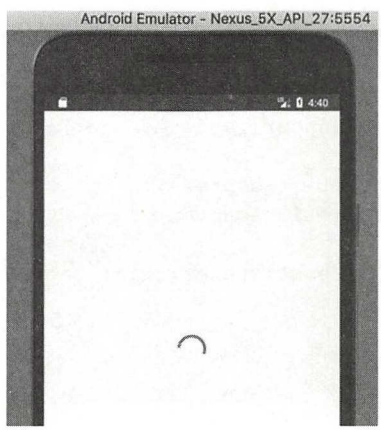


图 4-14　加载进度条的效果

通过观察可知，无缓存页面的长时间白屏，将会严重地降低用户体验，下面接着看看缓存模式的效果。

4.3.6 缓存模式

在缓存模式中，引入数据库的本地源，将网络数据保存至本地，显示数据都来源于数据库，网络只负责写入数据库，通过本地数据库保证数据的一致性。

在缓存页面 CacheActivity 中，与非缓存页面类似，额外注入本地数据库源 ObservableRepoDb 和添加下拉刷新控件 SwipeRefreshLayout。加载数据的逻辑位于 fetchUpdates()方法中，当下拉页面时，也会触发数据更新。

```
public class CacheActivity extends Activity {
    @Bind(R.id.cache_rv_list) RecyclerView mRvList; // 列表
    @Bind(R.id.cache_srl_swipe) SwipeRefreshLayout mSrlSwipe; // 刷新

    @Inject Application mApplication;
    @Inject ObservableRepoDb mRepoDb;
    @Inject GitHubClient mGitHubClient;

    private ListAdapter mListAdapter; //
    @Override protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_cache);
        ButterKnife.bind(this);

        ((RcApplication) getApplication()).getApiComponent().inject(this);

        LinearLayoutManager layoutManager = new LinearLayoutManager (mApplication);
        mRvList.setLayoutManager(layoutManager);

        mListAdapter = new ListAdapter();
        mRvList.setAdapter(mListAdapter);

        mSrlSwipe.setOnRefreshListener(this::fetchUpdates);
    }

    // ...
}
```

在缓存页面 CacheActivity 显示之前，即 onResume()，首先，使用异步的本地数据库源 mRepoDb，通过 setData()设置页面数据；其次，调用 fetchUpdates()获取网络数据，调用 mRepoDb 的 insertRepoList()方法，将数据插入至数据库。同时，在 insertRepoList()中，将会自动调用数据库 mRepoDb 的 getObservable()方法，通过 setData()设置页面数据。这样就会完成同步及缓存的全部逻辑。

```
@Override protected void onResume() {
    super.onResume();
    mRepoDb.getObservable()
```

```
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(this::setData);

        fetchUpdates();
        Toast.makeText(mApplication, "正在更新", Toast.LENGTH_SHORT).show();
    }

    // 设置数据，更新完成会调用
    private void setData(ArrayList<Repo> repos) {
        mListAdapter.setRepos(repos);
        Toast.makeText(mApplication, "更新完成", Toast.LENGTH_SHORT).show();
    }

    private void fetchUpdates() {
        // 延迟 3 秒，模拟网络较差的效果
        mGitHubClient.getRepos("SpikeKing")
                .delay(3, TimeUnit.SECONDS)
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(mRepoDb::insertRepoList,    this::fetchError,    this::
fetchComplete);
    }

    private void fetchError(Throwable throwable) {
        mSrlSwipe.setRefreshing(false);
    }

    private void fetchComplete() {
        mSrlSwipe.setRefreshing(false);
    }
```

最终的效果是在含有缓存的页面中，非初次启动时，都会优先加载数据库中的数据，向用户展示已存的数据，而异步的网络请求数据则用于更新数据库，当数据库的数据修改时，再将页面的显示更新至最新的数据。这样，用户在断网或者网络情况较差时，都可以看到页面的内容。

已加载数据的页面效果，如图 4-15 所示。

网络数据同步及缓存框架非常适合于新闻类的应用，当用户不希望在外面使用流量时，也可以阅读存入数据库中的新闻，当用户连接网络时，再将新闻的内容更新至最新。响应式编程框架非常适合于异步的网络请求，用很少的代码实现复杂的逻辑，同时，模块之间完全解耦，无论修改类中的任何一个逻辑，都不至于大范围地修改其他类。Retrofit 的核心功能在于处理网络请求，RxJava 的核心功能在于处理异步数据，Dagger 的核心功能在于解耦类的实例化。这三者是开发和测试异步请求数据的利器，受到大量 Android 开发者的喜爱，框架经过多次迭代，

已经日臻完善，是技术较强的公司的必备架构。对于高级的 Android 研发工程师，不仅需要学习大量的实用技术，还要学会如何将不同的技术进行组合，优雅地开发需求，最大限度地提升用户体验。

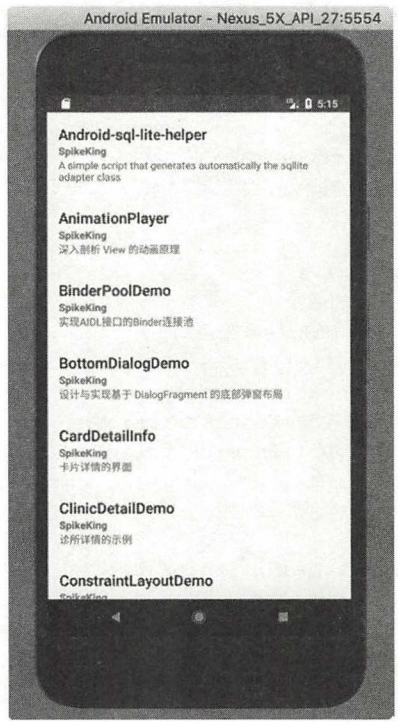


图 4-15　已加载数据的页面效果

第5章 炫酷功能

5.1 设计与实现朋友圈视频的滚动播放功能

在Android开发中，视频开发一直是比较复杂的概念。除了要熟悉Android系统相关的API，还需要掌握若干流媒体的知识，而视频又是交互不可缺少的一个环节。从文字到语音，再到视频，表述信息的语义会越来越丰富，更容易获得较好的用户体验。在和分享相关的应用中，通常需要支持用户分享视频，并在信息流中展示。有一种常见的需求是根据用户的滑动操作，自动并连续地播放与切换视频。

对于这种滚动播放功能的开发逻辑，一般是在页面中根据每个视频项的可视比例，找出露出最大的视频项，开始开始播放，同时停止播放其余视频项；当用户上下滑动屏幕时，自动切换与控制视频的播放状态与停止状态。视频开发的常用方式是基于 MediaPlayer，但是由于MediaPlayer是基础类库，功能较为简单，无法同步控制视频的播放和停止，无法满足这种连续播放视频的需求。

那么，应该如何优雅地完成这一需求呢？本节将通过一个实例，在普通视频播放的基础上，实现这样的滚动播放功能，讲述一些更高级的 Android 开发技巧。本例的视频数据源来自于网络。

本文实例的完整代码的下载地址为 https://github.com/SpikeKing/wcl-video-list-demo。

5.1.1 项目框架

在项目开发中，一般都需要依赖多个第三方开源库，这些开源库可以加快开发的速度，提高代码的稳定性。开源库一般包含三类，第一类是 Android 开发组提供的一些系统补充库，第二类是经过普遍认可的常用开源框架，第三类是针对特定功能的特定开源库。在项目的 build.gradle 中，第三方开源库位于 dependencies 中。本例同样包含这三个类别的开源库。

本例的系统补充库，即 RecyclerView 和 CardView，都属于 com.android.support 组。RecyclerView 是列表控件，比 ListView 拥有更多的定制模块，对于资源回收进行强制优化；CardView 是卡片控件，拥有圆角和阴影的视觉效果。

```
compile 'com.android.support:recyclerview-v7:23.+'
compile 'com.android.support:cardview-v7:23.+'
```

本例的常用开源框架，即 ButterKnife 和 Picasso，分别属于 com.jakewharton 组、com.squareup.picasso 组。ButterKnife 是布局导入库，通过特定的插件自动生成布局 ID 的绑定，如插件 Android ButterKnife Zelezny；Picasso 是图片加载库，将图片从网络中下载，设置图片控件（ImageView）并提供缓存，以简化加载网络图片的流程。

```
compile 'com.jakewharton:butterknife:7.0.1' // 依赖注入
compile 'com.squareup.picasso:picasso:2.5.2' // 图片加载
```

本例的特定开源库，即视频管理库，都是来源于 com.github.danylovolokh 组，分为 video-player-manager 和 list-visibility-utils。video-player-manager 是视频播放器的管理器，用于管理视频状态（开始、停止）的动态切换；list-visibility-utils 是列表可视化工具，包含一些管理视频可视化的常用方法。

```
compile 'com.github.danylovolokh:video-player-manager:0.2.0'
compile 'com.github.danylovolokh:list-visibility-utils:0.2.0'
```

其余的项目框架和 Gradle 配置与其他项目类似。

本例使用常见的 Activity + Fragment 的展现形式，核心逻辑位于 Fragment 中，Activity 仅用于加载 Fragment 和设置模式。Activity 使用默认的 MainActivity，首先，设置页面的导航栏，即 setSupportActionBar()，并且修改标题，调用 Toolbar 的 setTitle()方法；其次，加载 Fragment 页面，替换布局 ID main_fl_container 为 VideoListFragment。视频列表 Fragment 显示的视频包含两种模式，一种是本地（LOCAL）视频，一种是在线（ONLINE）视频，通过参数设置不同的模式。

```
public static final int LOCAL = 0; // 本地
public static final int ONLINE = 1; // 在线

@Bind(R.id.main_t_toolbar) Toolbar mTToolbar;
private FragmentManager mFM;
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ButterKnife.bind(this);

        mTToolbar.setTitle("播放列表");
        setSupportActionBar(mTToolbar);

        mFM = getSupportFragmentManager();
        if (savedInstanceState == null) {
            mFM.beginTransaction()
                    .replace(R.id.main_fl_container,
VideoListFragment.newInstance(LOCAL))
                    .commit();
        }
    }
```

在 MainActivity 的菜单选项中，支持切换视频列表的视频源，即本地视频与在线视频，通过设置不同的 VideoListFragment 参数，LOCAL 或 ONLINE，进行视频源的修改，在 MenuItem 的 setChecked()方法中，每次切换相反的状态（!item.isChecked()），实现视频源切换的目的。

```
    @Override public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.menu_main, menu);
        return true;
    }

    @Override public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
            case R.id.enable_local_video:
                if (!item.isChecked())
                    mFM.beginTransaction()
                            .replace(R.id.main_fl_container,
VideoListFragment.newInstance(LOCAL))
                            .commit();
                break;
            case R.id.enable_online_video:
                if (!item.isChecked())
                    mFM.beginTransaction()
                            .replace(R.id.main_fl_container,
VideoListFragment.newInstance(ONLINE))
                            .commit();
                break;
        }
        item.setChecked(!item.isChecked()); // 改变视频选中状态
        return true;
    }
```

页面布局采用非常简洁的 Toolbar 与 FrameLayout 的组合，Toolbar 是传统的 ActionBar 导航栏，用于显示标题与菜单等；FrameLayout 用于填充 Fragment 页面，显示真正的页面逻辑。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="org.wangchenlong.wcl_video_list_demo.MainActivity">

    <android.support.v7.widget.Toolbar
        android:id="@+id/main_t_toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        android:background="@color/colorPrimary"/>

    <FrameLayout
        android:id="@+id/main_fl_container"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
</LinearLayout>
```

最终效果是带有菜单的 Activity，如图 5-1 所示。

图 5-1　带有菜单的 Activity

5.1.2　视频列表

如果实现列表中视频的自动播放功能，需要创建一个视频列表。视频列表的逻辑在 VideoListFragment 中，下面会逐一进行讲解。

VideoListFragment 使用工厂模式 newInstance()方法创建对象，通过不同的视频类型参数，

创建不同的页面，将参数存入 VIDEO_TYPE_ARG 变量中，LOCAL 表示本地模式，ONLINE 表示线上模式。

```
public static VideoListFragment newInstance(int type) {
    VideoListFragment simpleFragment = new VideoListFragment();
    Bundle args = new Bundle();
    args.putInt(VIDEO_TYPE_ARG, type);
    simpleFragment.setArguments(args);
    return simpleFragment;
}
```

VideoListFragment 的构造器负责初始化若干参数。mList 是视频数据列表，每一项代表一个视频，类型是本地视频或线上视频；mVisibilityCalculator 是列表可视计算器，用于判断列表中的视频哪个是最佳播放项；mVideoPlayerManager 是视频播放管理器，用于管理视频的播放状态。

```
public VideoListFragment() {
    mList = new ArrayList<>();
    mVisibilityCalculator = new SingleListViewItemActiveCalculator(
            new DefaultSingleItemCalculatorCallback(), mList);
    mVideoPlayerManager = new SingleVideoPlayerManager(new PlayerItemChangeListener() {
        @Override
        public void onPlayerItemChanged(MetaData metaData) {
        }
    });

    mScrollState = AbsListView.OnScrollListener.SCROLL_STATE_IDLE;
    // 暂停滚动状态
}
```

在 onCreateView()方法中，VideoListFragment 填充布局并将布局绑定至 ButterKnife，fragment_video_list 布局非常简单，只有一个 RecyclerView 列表，用于显示连续的视频列表。

```
@Nullable @Override
public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container,
                         @Nullable Bundle savedInstanceState) {
    View view = LayoutInflater.from(getActivity())
            .inflate(R.layout.fragment_video_list, container, false);
    ButterKnife.bind(this, view);
    return view;
}
```

VideoListFragment 的核心逻辑位于 onViewCreated()方法中，主要含有三个部分，第一个部分是根据参数设置数据源，默认使用本地的数据源，可选使用线上的数据源(即网络视频地址)；第二个部分是设置 RecyclerList，setHasFixedSize()将 RecyclerView 的尺寸设置为固定，避免重绘，setLayoutManager()将 RecyclerView 的滚动方向设置为竖直，setAdapter()将 RecyclerView 的适配器设置为 VideoListAdapter；第三部分是滚动播放的重点，下面会详细讲解。

对于滚动播放，首先，通过 RecyclerViewItemPositionGetter 的实例封装 LinearLayoutManager 和 RecyclerView，提供获取列表项信息的接口；然后，在 RecyclerView 中添加滚动监听（addOnScrollListener），实现修改滚动状态的接口 onScrollStateChanged()和正在滚动的接口 onScrolled()。在修改滚动状态的接口中，即 onScrollStateChanged()，列表状态已经完成变化，如果状态是 SCROLL_STATE_IDLE，就表示滚动完成，列表处于静止状态，通过 mVisibilityCalculator 的 onScrollStateIdle 接口，输入 mItemsPositionGetter、列表的第一个和最后一个可视位置，判断哪个视频项被激活，已激活的视频项开始播放，其余的视频项停止播放。在正在滚动的接口中，即 onScrolled()，列表正在滚动，通过 mVisibilityCalculator 的 onScroll 接口，输入 mItemsPositionGetter、列表的第一个位置、列表的长度、上次的滚动状态，实时计算视频项所占页面的比例，动态播放或停止。

```
    @Override public void onViewCreated(View view, @Nullable Bundle savedInstanceState) {
        super.onViewCreated(view, savedInstanceState);

        initLocalVideoList();
        Bundle args = getArguments();
        if (args != null && args.getInt(VIDEO_TYPE_ARG) == MainActivity.ONLINE) {
            initOnlineVideoList();
        }

        mRvList.setHasFixedSize(true);
        mLayoutManager = new LinearLayoutManager(getActivity());
        mRvList.setLayoutManager(mLayoutManager);
        VideoListAdapter adapter = new VideoListAdapter(mList);
        mRvList.setAdapter(adapter);

        // 获取 Item 的位置
        mItemsPositionGetter = new RecyclerViewItemPositionGetter(mLayoutManager,
mRvList);
        mRvList.addOnScrollListener(new RecyclerView.OnScrollListener() {
            @Override
            public void onScrollStateChanged(RecyclerView recyclerView, int
scrollState) {
                mScrollState = scrollState;
                if (scrollState == RecyclerView.SCROLL_STATE_IDLE && !mList.isEmpty()) {
                    mVisibilityCalculator.onScrollStateIdle(
                            mItemsPositionGetter,
                            mLayoutManager.findFirstVisibleItemPosition(),
                            mLayoutManager.findLastVisibleItemPosition());
                }
            }

            @Override
            public void onScrolled(RecyclerView recyclerView, int dx, int dy) {
                if (!mList.isEmpty()) {
```

```
                mVisibilityCalculator.onScroll(
                        mItemsPositionGetter,
                        mLayoutManager.findFirstVisibleItemPosition(),
                        mLayoutManager.findLastVisibleItemPosition() -
                                mLayoutManager.findFirstVisibleItemPosition() + 1,
                        mScrollState);
            }
        }
    });
}
```

当启动页面时，即 onResume()，在列表的线程中，可视计算器根据滚动的状态，播放露出比例最大的视频，随着页面的启动，视频自动开始播放；当页面关闭时，即 onStop()，重置视频播放管理，暂停全部播放，准备下一次启动。

```
@Override public void onResume() {
    super.onResume();
    if (!mList.isEmpty()) {
        mRvList.post(new Runnable() {
            @Override
            public void run() {
                // 判断一些滚动状态
                mVisibilityCalculator.onScrollStateIdle(
                        mItemsPositionGetter,
                        mLayoutManager.findFirstVisibleItemPosition(),
                        mLayoutManager.findLastVisibleItemPosition());

            }
        });
    }
}

@Override
public void onStop() {
    super.onStop();
    mVideoPlayerManager.resetMediaPlayer(); //不显示页面时，释放播放器
}
```

其余的逻辑是初始化本地视频资源与网络视频资源，比较简单，具体逻辑可以参考源码。视频列表效果如图 5-2 所示。

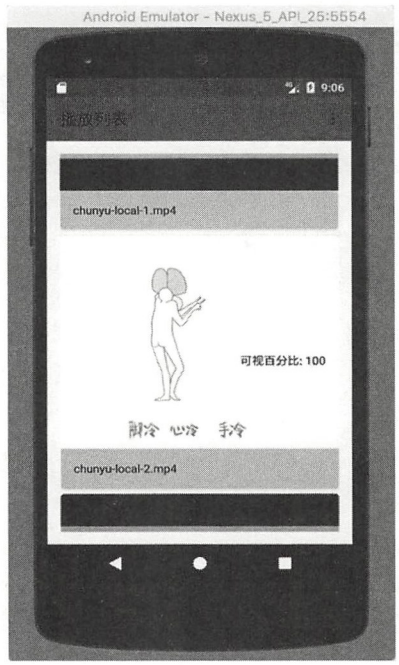


图 5-2　视频列表效果

5.1.3　视频项的适配器

RecyclerView 的本质是一个列表，通过适配器将数据加载至每一行，其中必须强制使用 ViewHolder 的模式，重复利用布局，优化加载速度。本例的适配器类是 VideoListAdapter，包含两个部分，第一部分是常规的覆写接口，供列表类调用，第二部分是一个 ViewHolder 的内部类，负责填充每一个视频项。

第一部分主要是覆写父类的接口，满足 RecyclerView 的调用，包含 onCreateViewHolder()、onBindViewHolder()、getItemCount()。首先，在构造器中设置视频数据源的列表；其次，在 onCreateViewHolder()中创建 ViewHolder，将 ViewHolder 绑定到每一项的视图 View 中，便于复用，优化加载速度，同时返回；接着，在 onBindViewHolder()中，将单个视频数据源绑定至视图的复用 ViewHolder，渲染列表的单项；最后，在 getItemCount()中，返回视频源的数量。

```
private final List<VideoListItem> mList; // 视频项列表

public VideoListAdapter(List<VideoListItem> list) {
    mList = list; // 数据源
}
```

```
    @Override
    public VideoListAdapter.VideoViewHolder onCreateViewHolder(ViewGroup parent,
int viewType) {
        View view = LayoutInflater.from(parent.getContext()).inflate(R.layout.
item_video, parent, false);
        VideoListAdapter.VideoViewHolder holder = new VideoListAdapter.
VideoViewHolder(view);
        view.setTag(holder); // 设置 Tag，在视频播放时使用，优化加载速度
        return holder;
    }

    @Override
    public void onBindViewHolder(final VideoListAdapter.VideoViewHolder holder, int
position) {
        VideoListItem videoItem = mList.get(position);
        holder.bindTo(videoItem);
    }

    @Override public int getItemCount() {
        return mList.size();
    }
```

第二部分主要是位于适配器中的 VideoViewHolder，用于填充列表中的项。核心是第三方库中的视频播放视图与监听，即 VideoPlayerView 和 MainThreadMediaPlayerListener。首先，使用 ButterKnife 绑定布局 ID 与视图；其次，创建播放视图的监听，在视频准备的接口（onVideoPreparedMainThread）中，隐藏默认的前置图片，在视频停止的接口（onVideoStoppedMainThread）中，显示默认的前置图片，给予用户良好的停播体验，并且将播放视图与监听绑定；接着，提供绑定接口，在接口中设置标题并使用 Picasso 绑定前置图片资源；最后，提供视频播放视图和百分比视图的访问接口。因为在 onCreateViewHolder()方法中，已经将 ViewHolder 绑定至视图的 Tag，所以，通过视图就可以访问 ViewHolder，同时设置 ViewHolder 的属性，即视频播放视图和百分比视图。

```
    public static class VideoViewHolder extends RecyclerView.ViewHolder {
        @Bind(R.id.item_video_vpv_player) VideoPlayerView mVpvPlayer; // 播放控件
        @Bind(R.id.item_video_iv_cover) ImageView mIvCover; // 覆盖层
        @Bind(R.id.item_video_tv_title) TextView mTvTitle; // 标题
        @Bind(R.id.item_video_tv_percents) TextView mTvPercents; // 百分比

        private Context mContext;
        private MediaPlayerWrapper.MainThreadMediaPlayerListener mPlayerListener;

        public VideoViewHolder(View itemView) {
            super(itemView);
            ButterKnife.bind(this, itemView);
            mContext = itemView.getContext().getApplicationContext();
```

```
        mPlayerListener = new MediaPlayerWrapper.MainThreadMediaPlayerListener()
{
            @Override public void onVideoSizeChangedMainThread(int width, int
height) {}
            @Override public void onVideoCompletionMainThread() {}
            @Override public void onErrorMainThread(int what, int extra) {}
            @Override public void onBufferingUpdateMainThread(int percent) {}

            @Override public void onVideoPreparedMainThread() {
                mIvCover.setVisibility(View.INVISIBLE); // 视频播放隐藏前图
            }

            @Override public void onVideoStoppedMainThread() {
                mIvCover.setVisibility(View.VISIBLE); // 视频暂停显示前图
            }
        };

        mVpvPlayer.addMediaPlayerListener(mPlayerListener);
    }

    public void bindTo(VideoListItem vli) {
        mTvTitle.setText(vli.getTitle());
        mIvCover.setVisibility(View.VISIBLE);
        Picasso.with(mContext).load(vli.getImageResource()).into(mIvCover);
    }

    public VideoPlayerView getVpvPlayer() { // 返回播放器
        return mVpvPlayer;
    }

    public TextView getTvPercents() { // 返回百分比视图
        return mTvPercents;
    }
}
```

视频项的布局也比较简单。最外层使用圆角的 CardView，内部主要包含 VideoPlayerView（视频播放器视图）、ImageView（停止播放的背景图片）、TextView（视频标题）、TextView（视频占整个页面的百分比，确定哪个视频用于播放）四个部分。

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.CardView
    app:cardCornerRadius="4dp">
    <RelativeLayout>
        <!--播放器-->
        <com.volokh.danylo.video_player_manager.ui.VideoPlayerView
            android:id="@+id/item_video_vpv_player"/>

        <!--背景-->
```

```
        <ImageView
            android:id="@+id/item_video_iv_cover"/>

        <!--标题-->
        <TextView
            android:id="@+id/item_video_tv_title">

        <!--百分比显示-->
        <TextView
            android:id="@+id/item_video_tv_percents"/>
    </RelativeLayout>
</android.support.v7.widget.CardView>
```

视频项效果如图 5-3 所示。

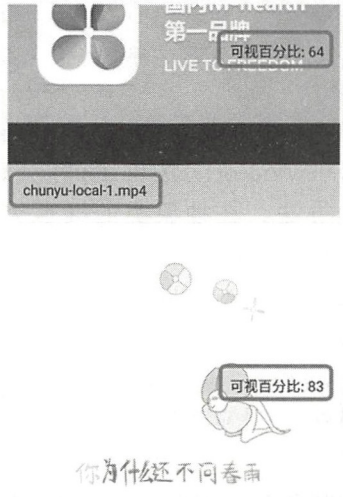


图 5-3　视频项效果

5.1.4　视频列表项

视频列表项即抽象类 VideoListItem，将视频资源与控制视频播放的逻辑融合在一起，将多个视频播放项组合成列表，列表作为参数传入 mVisibilityCalculator 和 VideoListAdapter 中，分别用于根据视频的屏幕占比选择可播放的视频，并显示视频的若干信息，如标题等。

视频列表项 VideoListItem 是一个抽象类，继承于接口 VideoItem 和 ListItem，分别用于视频的播放逻辑显示逻辑。VideoListItem 的构造器需要传递三个参数，视频播放器管理器 VideoPlayerManager 用于视频的播放管理，标题 title 用于存储视频的标题并且提供访问接口，视频停止的前置图片 imageResource 用于当停止播放视频时，视图显示的默认图片，也提供访

问接口。

```
    // 构造器，输入视频播放管理器
    public VideoListItem(VideoPlayerManager<MetaData> videoPlayerManager, String
title,
                         @DrawableRes int imageResource) {
        mVideoPlayerManager = videoPlayerManager;
        mTitle = title;
        mImageResource = imageResource;

        mCurrentViewRect = new Rect();
    }

    public String getTitle() {  // 视频项的标题
        return mTitle;
    }

    public int getImageResource() { // 视频项的背景
        return mImageResource;
    }
```

VideoItem 接口需要实现 playNewVideo()方法和 stopPlayback()方法，分别用于播放与停止，在抽象类中实现 stopPlayback()方法，在抽象类的继承类中实现 playNewVideo()方法。playNewVideo()方法就是抽象类 VideoListItem 的子类唯一需要实现的方法。stopPlayback()方法的逻辑是调用 VideoPlayerManager 的 stopAnyPlayback()方法，关闭全部正在播放的视频，具体实现逻辑位于 SingleVideoPlayerManager 类（继承于 VideoPlayerManager）中。

```
    @Override public void stopPlayback(VideoPlayerManager videoPlayerManager) {
        videoPlayerManager.stopAnyPlayback();
    }
```

ListItem 接口需要实现 getVisibilityPercents()方法、setActive()方法和 deactivate()方法，分别用于当前视频在屏幕中的露出比例、启动播放视频和停止播放视频。

- ◎ getVisibilityPercents()方法：参数是视频项的百分比视图。首先，获取视频项的位置信息，存入 mCurrentViewRect，再获取当前视图的高度，存入 height；其次，根据视频项的位置信息与当前视频的高度，计算视频在屏幕中露出的百分比；最后，将百分比信息显示在百分比视图（视频项的百分比 TextView）中，同时返回百分比信息。
- ◎ setActive()方法：参数是新的播放视图与这个视图的位置，第一步，在视频项的视图中，获取 ViewHolder；第二步，将参数打包成 CurrentItemMetaData，与 ViewHolder 的视频播放器视图（VideoPlayerView）和视频播放管理（VideoPlayerManager），组成三个参数，一起传入子类的 playNewVideo()方法中，

执行视频的播放。

◎ deactivate()方法：参数是当前的播放视图与该视图的位置，调用 stopPlayback()函数，关闭全部正在播放的视频。

```
    @Override public int getVisibilityPercents(View view) { // 显示可视的百分比程度
        int percents = 100;

        view.getLocalVisibleRect(mCurrentViewRect);
        int height = view.getHeight();

        if (viewIsPartiallyHiddenTop()) {
            percents = (height - mCurrentViewRect.top) * 100 / height;
        } else if (viewIsPartiallyHiddenBottom(height)) {
            percents = mCurrentViewRect.bottom * 100 / height;
        }
        // 设置百分比
        setVisibilityPercentsText(view, percents);

        return percents;
    }

    @Override public void setActive(View newActiveView, int newActiveViewPosition)
{
        VideoListAdapter.VideoViewHolder viewHolder =
                (VideoListAdapter.VideoViewHolder) newActiveView.getTag();
        playNewVideo(new                CurrentItemMetaData(newActiveViewPosition,
newActiveView),
                viewHolder.getVpvPlayer(), mVideoPlayerManager);
    }

    @Override public void deactivate(View currentView, int position) {
        stopPlayback(mVideoPlayerManager);
    }
```

抽象类 VideoListItem 的子类有两个，一个是本地视频项 LocalVideoListItem，另一个是网络视频项 OnlineVideoListItem，它们分别实现不同的 playNewVideo 接口。本地视频项 LocalVideoListItem 的构造器，除了输入父类的参数外，额外输入本地的文件资源，存储在 Assest 目录下。在 playNewVideo()方法中，调用 VideoPlayerManager 的 playNewVideo()函数，播放本地的视频资源。

```
    public class LocalVideoListItem extends VideoListItem {
        private final AssetFileDescriptor mAssetFileDescriptor; // 资源文件描述

        public LocalVideoListItem(VideoPlayerManager<MetaData> videoPlayerManager,
String title,
                                  @DrawableRes int imageResource, AssetFileDescriptor
assetFileDescriptor) {
```

```
            super(videoPlayerManager, title, imageResource);
            mAssetFileDescriptor = assetFileDescriptor;
        }

        @Override
        public void playNewVideo(MetaData currentItemMetaData, VideoPlayerView
player,
                                 VideoPlayerManager<MetaData> videoPlayerManager) {
            videoPlayerManager.playNewVideo(currentItemMetaData, player, mAssetFile
Descriptor);
        }
    }
```

本地视频采用Assest的存储方式，视频位于src/main/assets的目录下，通过调用AssetManager的openFd方法，获取本地的视频资源。

```
    // 获取资源文件
    private AssetFileDescriptor getFile(String name) {
        try {
            return getActivity().getAssets().openFd(name);
        } catch (IOException e) {
            e.printStackTrace();
            return null;
        }
    }
```

Assets目录在项目中的位置如图5-4所示。

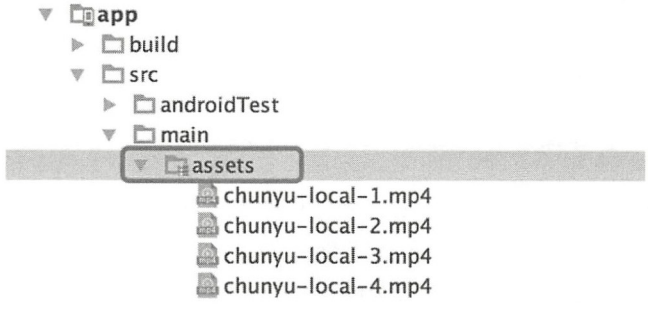


图5-4 Assets目录在项目中的位置

网络视频项OnlineVideoListItem的构造器，除了输入父类的参数外，额外输入网络视频地址，即HTTP或HTTPS的网址。同样地，在playNewVideo()方法中，调用VideoPlayerManager的playNewVideo()函数，播放网络的在线视频资源。

```
    public class OnlineVideoListItem extends VideoListItem {
        private final String mOnlineUrl; // 资源文件描述
```

```
        public OnlineVideoListItem(VideoPlayerManager<MetaData> videoPlayerManager,
String title,
                                   @DrawableRes int imageResource, String onlineUrl) {
            super(videoPlayerManager, title, imageResource);
            mOnlineUrl = onlineUrl;
        }

        @Override
        public void playNewVideo(MetaData currentItemMetaData, VideoPlayerView
player, VideoPlayerManager<MetaData> videoPlayerManager) {
            videoPlayerManager.playNewVideo(currentItemMetaData, player, mOnlineUrl);
        }
    }
```

视频的滚动播放功能属于高级功能，在原有的播放视频功能中，添加列表的逻辑控制，动态播放露出最多的视频项。在开发中，根据功能的需求与特点，动态设计开发流程。

- ◎ 导入开源的第三方库，除了常用的系统库和加速开发库以外，还需要根据需求，寻找最合适的第三方库加以修改，并满足当前的需要。既要保证功能的完整，又要保证代码的质量。
- ◎ 页面架构最好使用 Activity & Fragment 的模式，便于替换，将控制与逻辑解耦，页面控制在 Activity 中，页面逻辑在 Fragment 中。
- ◎ 熟练使用列表控件 RecyclerView，代替 ListView，理解 RecyclerView 中的接口含义，如设置列表的滚动监听、获取列表中的可视项位置等，还要理解 RecyclerView 的 ViewHolder 模式。
- ◎ 熟练使用第三方库的类与方法，如视频播放器类 VideoPlayerManager、视频播放视图 VideoPlayerView、视频项类 VideoListItem，并将这些类与方法关联起来，完成特定功能。

将多个视频放入列表中播放非常容易导致内存泄漏，在开发的时候一定注意。每个视频都需要在异步线程中播放，同时只能有一个视频处于播放状态，当停止其他视频播放时，需要及时释放线程占用的资源，并且当关闭页面时，需要及时释放所有线程占用的资源。总的来说，视频的滚动播放功能还是比较炫酷的，涉及的知识点也较多，希望读者在学习中，根据源码和讲解多多练习，只有不断开发复杂的功能，才能得到最快的技术提升。

5.2　设计与实现基于 DialogFragment 的底部弹窗布局

在 Android API Level 11 中，DialogFragment 被加入到基础组件库，是一类较为特殊的 Fragment，用于在 Activity 的窗口之上添加对话框，其典型的应用在于警告框、输入框、确认框

等。DialogFragment 包含一个 Dialog 对象，展示的外观基于 Fragment 的样式，通过已封装的接口控制 Dialog 对象的显示或隐藏状态，如图 5-5 所示。

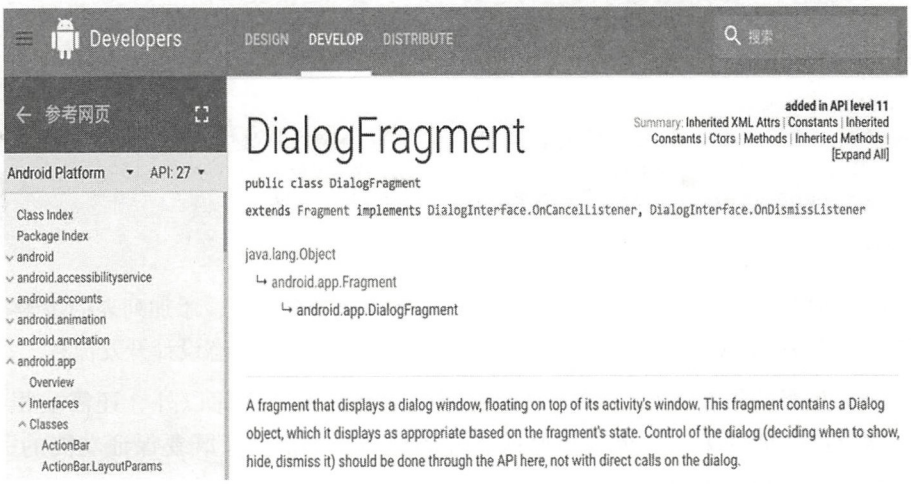


图 5-5　DialogFragment

本例根据 DialogFragment 的属性开发出一种特殊的样式，非常优雅和简洁地实现底部弹窗功能。虽然底部弹窗与普通的 Dialog 不同，布局需要紧贴页面的下部，但本质上仍属于 Fragment 布局，通过继承 DialogFragment 类，实现不同接口，即可定制不同样式的 Fragment。除了设计不同的布局之外，还需要考虑 Fragment 的主题样式（Theme Style），完美地实现底部弹窗效果。

本文实例完整代码的下载地址为 https://github.com/SpikeKing/BottomDialogDemo。

5.2.1　首页逻辑

本例依据一个非常基础的 HelloWorld 工程，完整地展示如何使用 DialogFragment 的全部技巧，工程配置保留默认。在项目依赖中，添加 ButterKnife 第三库，简化处理布局文件与页面类之间的关系，加快开发速度。

```
compile 'com.jakewharton:butterknife:8.1.0'
apt 'com.jakewharton:butterknife-compiler:8.1.0'
```

在主页 MainActivity 中，除了常规地设置布局 setContentView()之外，仅提供一个展示底部弹出的接口 showBottomDialog()。由于底部弹窗 BottomDialogFragment 本质仍属于 Fragment，因此需要依赖 FragmentManager 进行展示。调用 DialogFragment 的 show()方法即可显示 Dialog。底部弹窗的核心逻辑位于 BottomDialogFragment 中。

```
public class MainActivity extends AppCompatActivity {
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    // 显示底部 Dialog
    public void showBottomDialog(View view) {
        FragmentManager fm = getSupportFragmentManager();
        BottomDialogFragment editNameDialog = new BottomDialogFragment();
        editNameDialog.show(fm, "fragment_bottom_dialog");
    }
}
```

如图 5-6 所示，首页布局非常简洁，仅含有一个展示底部弹窗的按钮，即 Bottom Dialog。点击按钮，调用 showBottomDialog()方法，触发显示底部弹窗 BottomDialogFragment。

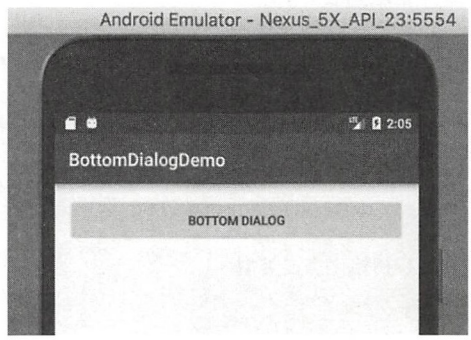


图 5-6　MainActivity

5.2.2　弹窗样式

在底部弹窗 BottomDialogFragment 类中，覆写 onCreateDialog()方法，创建 Dialog 类。onCreateDialog()是 DialogFragment 的入口方法，类似于 Activity 的 onCreate()方法。

在 onCreateDialog()方法中，实例化 Dialog 类并设置属性：

（1）创建 Dialog 实例，参数是 Context 和自定义样式 BottomDialog。

（2）设置 Dialog 的窗口属性（Window），不含标题，即 Window.FEATURE_NO_TITLE。

（3）设置 Dialog 的布局资源，调用 setContentView()。

（4）开启 Dialog 的布局外点击触发会话关闭功能，调用 setCanceledOnTouchOutside()。

（5）重置 Dialog 的窗口布局，中心为紧贴底部，即 Gravity.BOTTOM，宽度为屏幕宽度，

即 LayoutParams.MATCH_PARENT。

接着，使用 ButterKnife 将 BottomDialogFragment 类绑定至 Dialog，通过注解@Bind 访问布局 ID，在 ButterKnife#bind()方法中，本应绑定视图（View），因为布局（Layout）被设置于 Dialog 中，Dialog 也继承于视图（View），所以绑定 Dialog。同时，在 initClickTypes()方法中，设置布局中的点击事件逻辑。

```
@NonNull @Override
public Dialog onCreateDialog(Bundle savedInstanceState) {
    // 使用不带 Theme 的构造器，获得的 dialog 边框距离屏幕仍有几毫米的缝隙。
    Dialog dialog = new Dialog(getActivity(), R.style.BottomDialog);

    dialog.requestWindowFeature(Window.FEATURE_NO_TITLE);
    dialog.setContentView(R.layout.fragment_bottom);
    dialog.setCanceledOnTouchOutside(true); // 外部点击取消

    // 设置宽度为屏宽，靠近屏幕底部。
    Window window = dialog.getWindow();
    WindowManager.LayoutParams lp = window.getAttributes();
    lp.gravity = Gravity.BOTTOM; // 紧贴底部
    lp.width = WindowManager.LayoutParams.MATCH_PARENT; // 宽度持平
    window.setAttributes(lp);

    ButterKnife.bind(this, dialog); // Dialog 即 View

    initClickTypes();  // 初始化点击类型

    return dialog;
}
```

Dialog 底部弹窗的样式（Dialog）与默认的样式不同，因此在 Dialog 中重新设置样式，样式属性位于 style 文件的 BottomDialog 属性中。

（1）layout_width：Dialog 的水平宽度填充屏幕，即属性值 match_parent。

（2）wrap_content：Dialog 的竖直高度与布局高度一致，即属性值 wrap_content。

（3）windowIsFloating：Dialog 悬浮于窗口之上，即属性值 true。

（4）backgroundDimEnabled：Dialog 背景设置为透明，取消暗色，即属性值 false。

```
<style name="BottomDialog" parent="@style/AppTheme">
    <item name="android:layout_width">match_parent</item>
    <item name="android:layout_height">wrap_content</item>
    <item name="android:windowIsFloating">true</item>
    <item name="android:backgroundDimEnabled">false</item>
</style>
```

这样，底部弹窗 BottomDialogFragment 紧贴于屏幕下方，样式整齐，如图 5-7 所示。

图 5-7　底部弹窗

5.2.3　弹窗逻辑

底部弹窗的逻辑位于 initClickTypes()方法中，包含三个部分：

（1）初始化控件组 initViewArray()，用于批量控制控件集合。

（2）设置布局点击事件 initLayout()，根据用户行为修改页面展示。

（3）发送按钮 mTvSend 的逻辑，当点击发送按钮时，根据当前选中状态执行不同的策略。

策略逻辑共分为四类：一字之师、妙语连珠、学识丰富、博学多才。根据策略逻辑的不同，动态地扣除金币，同时弹框（Toast）提示。

```
private void initClickTypes() {
    initViewArray(); // 初始化控件组
    initLayout(); // 设置布局点击事件

    mTvSend.setOnClickListener(new View.OnClickListener() {
        @Override public void onClick(View v) {
            String typeStr = "金币不足";
            switch (mType) {
                case 0:
                    if (countCoins(100))
                        typeStr = "一字之师";
                    break;
                case 1:
                    if (countCoins(200))
                        typeStr = "妙语连珠";
                    break;
                case 2:
                    if (countCoins(800))
                        typeStr = "学识丰富";
```

```
                    break;
                case 3:
                    if (countCoins(1200))
                        typeStr = "博学多才";
                    break;
                default:
                    break;
            }
            Toast.makeText(getContext(), typeStr, Toast.LENGTH_SHORT).show();
        }
    });
}
```

在初始化控件组 initViewArray()中，分别将四类布局容器（Layout）、文本视图（TextView）、图片视图（ImageView）放入列表中，以便统一管理。其中，布局为了接收点击事件，当用户选中策略时，文字文案变为白色，选中对号图片显示；其余策略，文字文案为灰色，选中对号图片隐藏。其中，对于视图类的命名规范，Tv 是 TextView 的缩写，Iv 是 ImageView 的缩写。

```
private void initViewArray() {
    mLayouts = new ArrayList<>();
    mTvTypes = new ArrayList<>();
    mIvTypes = new ArrayList<>();

    mLayouts.add(mLlFirstContainer);
    mLayouts.add(mLlSecondContainer);
    mLayouts.add(mLlThirdContainer);
    mLayouts.add(mLlForthContainer);

    mTvTypes.add(mTv100Coins);
    mTvTypes.add(mTv2Yuan);
    mTvTypes.add(mTv8Yuan);
    mTvTypes.add(mTv12Yuan);

    mIvTypes.add(mIv100Coins);
    mIvTypes.add(mIv2Yuan);
    mIvTypes.add(mIv8Yuan);
    mIvTypes.add(mIv12Yuan);
}
```

选中策略效果如图 5-8 所示。

图 5-8　选中策略

在设置布局点击事件 initLayout()中，首先调用 chooseRegardsType()方法，选中默认的类型 mType。接着，遍历布局数组 mLayouts，设置布局的监听事件 setOnClickListener，当触发事件时，调用 chooseRegardsType()，选中布局对应的策略类型 mType。变量 mType 用于保存当前选中的类型。

```
private int mType = 0; // 第一个类型

private void initLayout() {
    chooseRegardsType(mType); // 选择默认类型

    for (int i = 0; i < mLayouts.size(); i++) {
        final int tmp = i;
        LinearLayout ll = mLayouts.get(i);
        ll.setOnClickListener(new View.OnClickListener() {
            @Override public void onClick(View v) {
                mType = tmp;
                chooseRegardsType(mType);
            }
        });
    }
}
```

在选择策略 chooseRegardsType()方法中，根据选中的策略类型 type，将相应的文本视图和图片视图的 Enable 状态设置为 False，同时，将非选中的策略设置为 True，从而实现样式间的切换。

```
private void chooseRegardsType(int type) {
    int size = mTvTypes.size();
    for (int i = 0; i < size; ++i) {
        if (i != type) {
            mTvTypes.get(i).setEnabled(true);
            mIvTypes.get(i).setEnabled(true);
        } else {
            mTvTypes.get(i).setEnabled(false);
            mIvTypes.get(i).setEnabled(false);
        }
```

```
    }
}
```

通过切换视图的 Enble 状态，改变视图的样式。例如，将 TextView 文字颜色的属性 textColor 设置为 regard_text_bkg。

```
android:textColor="@color/regard_text_bkg"
```

在 regard_text_bkg 中，不是存储单一的颜色值，而是一个颜色值组。默认颜色是灰色，处于非选中状态，即 Enable 状态值为 true；反之，当 Enable 状态值为 false 时，处于选中状态，颜色是白色。

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:color="#848484" android:state_enabled="true"/>
    <item android:color="#FFFFFF" android:state_enabled="false"/>
</selector>
```

本例根据已有的 DialogFragment 类，定制布局样式，添加逻辑关系，使用 FragmentManager 控制并显示 Dialog，从而实现经典的底部弹窗样式。DialogFragment 是标准弹窗样式，在内部，将 Dialog 已有的接口进行封装，通过 Fragment 的特有属性，连接 Activity 与 Dialog，管理弹窗与页面的依赖，实现弹窗布局。本例正是利用这一特性，定制 Dialog 与 Fragment 的不同接口与样式，实现不同的底部弹窗布局。在开发中，也经常需要定制已有的组件，通过改变样式或者改变位置创建新的组件，使之满足我们的需求，为产品提供更丰富的选择。不要拘泥于已有的组件，多研究源码，深入理解内部细节的逻辑，才能改写这些逻辑。研究源码和创新的能力也是高级程序员必备的素养之一。

第6章 精美动画

6.1 实现页面切换中元素分享的动画效果

在 Android 应用中，除了完成必要的功能以外，也需要优美的页面展示。页面元素不仅需要精美的图标、协调的配色，也需要有一些合适的动画。通过动画效果，可以强调一些重点元素，给予用户更多的提示信息。例如，在应用的首页展示时，一般而言会直接显示页面的布局，这种方式简称为冷启动（Cold Start）。它没有任何感情色彩，如果需要强调可点击的位置，只能通过阴影、配色、用户常识等信息。当然，也可以添加一些控件动画，让首页展示更加有趣，更容易突出页面中的可点击元素，增加访问量，提升整体的体验，让用户更容易接受。

在页面元素的动画中，一般主要有两类，一类是效果显示模式，另一类是预留位置模式。

（1）效果显示模式。通过动画效果逐渐地显示控件，如位移、渐变、缩放等动画，也可以组合多个。

（2）预留位置模式。改变控件的位置和大小，直至与预设位置一致，如坍塌、滑入等动画。

本节会通过实例讲解如何实现这两类页面动画的基本样式，读者也可自行扩展，满足页面设计师的更多动画需求，为应用增加更多的趣味性和用户体验。

本实例完整代码的下载地址为 https://github.com/SpikeKing/wcl-onboarding-demo。

6.1.1 项目框架

本例作为一个实验项目，主要关注如何实现 Android 的动画效果，在 Gradle 配置中，添加

一些常用的工具库，加快开发的速度。本例使用四个常见的、同时非常成熟的第三方开源库，即 CardView、RecyclerView、Picasso、ButterKnife。CardView 是卡片样式的视图，外观含有圆角和阴影；RecyclerView 是列表样式的视图，与 ListView 类似，展示一组相同样式的项；Picasso 是图片加载库，用于加载网络或本地的图像资源；ButterKnife 是布局属性绑定库，更加便捷地在页面中绑定布局 ID。这些开源库不仅可以应用于实验项目，在实际开发中，也可以使用它们，以简化开发流程。

```
    compile 'com.android.support:cardview-v7:23.0.0'
    compile 'com.android.support:recyclerview-v7:23.0.0'
    compile 'com.squareup.picasso:picasso:2.5.2'
    compile 'com.jakewharton:butterknife:7.0.1'
```

同时，项目也集成 Java 1.8 版本的 Lambda 表达式，简化匿名类的编写。

```
    plugins {
        id "me.tatarka.retrolambda" version "3.2.5"
    }

    android {
        compileOptions {
            sourceCompatibility JavaVersion.VERSION_1_8
            targetCompatibility JavaVersion.VERSION_1_8
        }
    }
```

首页比较简单，包含两个按钮，即 mBCenter 和 mBPlaceholder，分别跳转至两个页面，即 CenterActivity 和 PlaceholderActivity，一个页面（CenterActivity）用于展示效果显示的动画效果，另一个页面（PlaceholderActivity）用于展示预留位置的动画效果。

```
    public class MainActivity extends AppCompatActivity {
        @Bind(R.id.main_b_onboard_center) Button mBOnboardCenter;
        @Bind(R.id.main_b_onboard_placeholder) Button mBOnboardPlaceholder;

        @Override
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main);
            ButterKnife.bind(this);

            mBOnboardCenter.setOnClickListener(v ->
                    startActivity(new Intent(MainActivity.this, OnboardingCenterActivity.
class)));
            mBOnboardPlaceholder.setOnClickListener(v ->
                    startActivity(new  Intent(MainActivity.this,  OnboardingPlaceholder
Activity.class)));
        }
    }
```

布局资源也比较简单，父布局是 RelativeLayout，子布局是两个 Button，分别用于跳转对应的页面。

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout>
    <Button
        android:id="@+id/main_b_center"
        android:text="效果显示"/>

    <Button
        android:id="@+id/main_b_placeholder"
        android:layout_below="@+id/main_b_center"
        android:text="预留位置"/>
</RelativeLayout>
```

最终，首页的效果如图 6-1 所示。

图 6-1　首页的效果

6.1.2　效果显示动画

在控件的显示过程中，除了直接静态显示，还可以使用动画显示。常用的三种动画样式是位移、渐变、缩放。位移表示控件的位置发生变化，渐变表示控件由透明变为实体，缩放表示控件的大小发生变化。这三种动画样式可以单独使用，也可以组合使用。效果显示动画模式的核心思想是将控件原有的静态显示转换为动态显示，通过位移、渐变、缩放等动画效果，突出页面中的核心元素。

本例的效果显示页面布局含用五个子控件元素，一个图片控件（ImageView）、两个文本控件（TextView）、两个按钮控件（Button）。父控件是帧布局（FrameLayout），含有一个竖直的线性布局（LinearLayout）和一个居中的图片控件，两个文本控件与两个按钮控件竖直排列于

LinearLayout 中。五个子控件元素全部使用动画的方式显示出来。

```
<FrameLayout>
    <LinearLayout
        android:layout_gravity="center"
        android:gravity="center"
        android:orientation="vertical">
        <TextView/>
        <TextView/>

        <Button
            android:id="@+id/onboard_b_choice_1"/>
        <Button
            android:id="@+id/onboard_b_choice_2"/>
    </LinearLayout>

    <ImageView
        android:id="@+id/onboard_iv_logo"
        android:layout_gravity="center"/>
</FrameLayout>
```

两个文本控件起始的透明度（alpha）属性设置为 0，即完全透明，为了在开发中显示最终的效果，tools 命名控件的透明度（alpha）属性设置为 1，即实体化。tools 命名空间的属性仅仅在开发中起作用，并不会影响实际的效果，用于测试。通过控件设置透明度属性可知，在本例中，两个 TextView 控件用于展示渐变动画。

```
<TextView
    android:alpha="0"
    android:text="Hello World!"
    tools:alpha="1"/>

<TextView
    android:alpha="0"
    android:text="My name is Spike."
    tools:alpha="1"/>
```

两个按钮控件起始的水平与竖直缩放（scaleX）属性设置为 0，即完全消失；同样地，tools 命名控件的透明度（alpha）属性设置为 1，即正常尺寸。通过控件设置缩放属性可知，在本例中，两个按钮控件用于展示缩放动画。

```
<Button
    android:id="@+id/onboard_b_choice_1"
    android:scaleX="0"
    android:scaleY="0"
    android:text="Talk to me"
    tools:scaleX="1"
    tools:scaleY="1"/>
```

```
<Button
    android:id="@+id/onboard_b_choice_2"
    android:scaleX="0"
    android:scaleY="0"
    android:text="Quit"
    tools:scaleX="1"
    tools:scaleY="1"/>
```

一个图片控件位于整个布局的中部，在本例中用于展示位移动画。

```
<ImageView
    android:id="@+id/onboard_iv_logo"
    android:layout_gravity="center"
    android:contentDescription="@null"
    android:src="@drawable/img_face"/>
```

布局部分已经介绍完成，接着讲解如何编写动画效果的逻辑，动画效果主要分为三个部分，位移、渐变和缩放，下面逐一进行讲解。

位移动画，图片控件的实例（mIvLogo）竖直向上移动，调用 ViewCompat 的属性设置动画的参数，animate()设置执行动画的控件，translationY()设置沿着 Y 轴移动的距离，setStartDelay()设置动画延迟开始的时间，setDuration()设置动画的持续时间，setInterpolator()设置移动速度的差值器，最后调用 start()方法，启动整个动画。动画效果是图片控件实例 mIvLogo，沿着 Y 轴水平向上移动 300 个像素，延迟 300 毫秒（STARTUP_DELAY 的值）开始，持续 1000 毫秒（ANIM_ITEM_DURATION 的值），移动的速度越来越慢（DecelerateInterpolator）。

```
// 向上移动
ViewCompat.animate(mIvLogo)
        .translationY(-300)
        .setStartDelay(STARTUP_DELAY)
        .setDuration(ANIM_ITEM_DURATION)
        .setInterpolator(new DecelerateInterpolator(1.2f))
        .start();
```

渐变动画与缩放动画，首先获取线性布局实例 mLlContainer 的所有元素，即两个按钮控件、两个文本控件。按钮控件用于渐变动画，文本控件用于缩放动画。通过实例的类型进行区分所遍历的子控件。

在按钮控件的动画中，同样是调用 ViewCompat 的属性设置动画的参数，核心是 alpha()设置透明度的值，与 translationY()联合使用。按钮控件的动画效果是沿着 Y 轴水平向下移动 50 个像素，透明度由全透明变为实体化（0~1），起始延迟 500 毫秒，其余逐个增加 300 毫秒（ITEM_DELAY），持续 1000 毫秒（ANIM_ITEM_DURATION 的值），移动的速度越来越慢（DecelerateInterpolator）。

在文本控件的动画中，与按钮控件类似，核心是 scaleY()设置竖直缩放比例和 scaleX()设置

水平缩放比例。按钮控件的动画效果就是在位置中逐渐显示，起始不可视，即缩放比例为 0，逐渐转变为原有尺寸（1），其余与按钮控件类似。

> 注意：使用 instanceof 区分按钮控件和文本控件时，一定要匹配按钮控件，因为按钮控件是文本控件的子类，如果匹配文本控件，则无论是按钮控件还是文本控件都会全部命中。

```
    for (int i = 0; i < mLlContainer.getChildCount(); i++) {
        View v = mLlContainer.getChildAt(i);
        ViewPropertyAnimatorCompat viewAnimator;
        //文本控件，按钮控件是文本控件的子类
        if (!(v instanceof Button)) {
            // 渐变动画，从消失到显示
            viewAnimator = ViewCompat.animate(v)
                    .translationY(50)
                    .alpha(1)
                    .setStartDelay((ITEM_DELAY * i) + 500)
                    .setDuration(ANIM_ITEM_DURATION);
        } else { // 按钮控件控件，从缩小到扩大
            viewAnimator = ViewCompat.animate(v)
                    .scaleY(1)
                    .scaleX(1)
                    .setStartDelay((ITEM_DELAY * i) + 500)
                    .setDuration(ANIM_ITEM_DURATION / 2);
        }
        viewAnimator.setInterpolator(new DecelerateInterpolator()).start();
    }
```

接着，在两个按钮中添加回调事件，一个按钮输出 Toast 提示信息，另一个按钮设置点击回退，即调用 onBackPressed()方法。

```
    mBChoice1.setOnClickListener(v ->
            Toast.makeText(getApplicationContext(),                    "Nice!",
Toast.LENGTH_SHORT).show());
    mBChoice2.setOnClickListener(v -> onBackPressed());
```

最终，效果显示动画的页面如图 6-2 所示。

图 6-2　效果显示动画

6.1.3　预留位置动画

预留位置动画是控件的位置已经预留，控件以动画的形式移动到预留的位置中。本例为了更好地展示这一动画效果的实现过程，使用三个组件，一个是顶部的工具栏（Toolbar），另一个是右下角的漂浮活动按钮（FloatingActionButton，Fab），最后一个是中部的列表（RecyclerView）项，动画效果分别是坍塌、渐入、滑动。下面将逐一地讲解这三个组件的动画实现方法及原理。

页面布局的根布局是 FrameLayout，包含竖直的 LinearLayout 和 FloatingActionButton。LinearLayout 含有两个控件，工具条 Toolbar 和列表 RecyclerView，并且在 Toolbar 中，含有一个 TextView，用于显示标题。

```
<FrameLayout>
    <LinearLayout
        android:orientation="vertical">
        <android.support.v7.widget.Toolbar
            android:id="@+id/placeholder_t_toolbar">

            <TextView
                android:id="@+id/placeholder_tv_title"
```

```
                android:alpha="0"/>
        </android.support.v7.widget.Toolbar>

        <android.support.v7.widget.RecyclerView
            android:id="@+id/placeholder_rv_recycler"/>
    </LinearLayout>

    <android.support.design.widget.FloatingActionButton
        android:id="@+id/placeholder_fab_bar"
        android:scaleX="0"
        android:scaleY="0"
        app:fabSize="normal"/>
</FrameLayout>
```

PlaceholderActivity 页面负责承载这三个动画控件。动画效果不是在页面启动时即刻发生，而是选择延迟 500 毫秒，即 0.5 秒，这样更能加深用户对于动画的感知。Handler()创建主线程的接口，调用 postDelayed()方法，延迟 500 毫秒执行 onAnimateCreate()方法，即延迟执行动画效果，注意本例使用 Lambda 的方法引用（Method References）模式，简化匿名类的编写。

```
@Override protected void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_onboarding_placeholder);
    ButterKnife.bind(this);

    new Handler().postDelayed(this::onAnimateCreate, 500); // 延迟启动动画效果
}
```

动画效果的实现逻辑，一部分是列表项的动画，即 RecyclerView 的 Item，通过 setItemAnimator()方法，将动画设置为淡入效果；另一部分是顶部的工具栏，添加 onPreDraw()方法的监听，在监听的接口中，首先删除其他的监听，避免重复，其次调用 collapseToolbar()方法，输入 Toolbar 的实际高度（getHeight()），设置 Toolbar 的坍塌效果。Toolbar 的具体动画逻辑在 collapseToolbar()方法中实现。注意第一行，使用 ViewCompat 的 animate()方法，将透明的 TextView（mTvTitle）变成非透明，此 TextView 位于 ToolBar 中。

```
private void onAnimateCreate() {
    ViewCompat.animate(mTvTitle).alpha(1).start();

    mRvRecycler.setLayoutManager(new LinearLayoutManager(this));
    mRvRecycler.setItemAnimator(ItemAnimatorFactory.slidein());  // 列表项的淡
入动画

    mPhRecyclerAdapter = new PhRecyclerAdapter();
    mRvRecycler.setAdapter(mPhRecyclerAdapter);

    mTToolbar.getViewTreeObserver().addOnPreDrawListener(new
ViewTreeObserver.OnPreDrawListener() {
```

```
            @Override public boolean onPreDraw() {
                mTToolbar.getViewTreeObserver().removeOnPreDrawListener(this);
                collapseToolbar(mTToolbar.getHeight());  // Toolbar 的坍塌效果
                return true;
            }
        });
    }
```

接着，继续讲解 collapseToolbar()的动画逻辑。参数是 height，即当前 ToolBar 的高度。创建 TypedValue；通过 resolveAttribute()获取 ActionBar 的高度；通过 complexToDimensionPixelSize()将高度转换为像素；最终的值存储在 toolBarHeight 中，动画效果的范围在 height 与 toolBarHeight 之间，即 Toolbar 的高度与 ActionBar 的高度之间。使用值动画 ValueAnimator，范围是 height 至 toolBarHeight；通过 ValueAnimator 的 addUpdateListener()接口监听值的变化过程；当值发生改变时，通过 ValueAnimator 的 getAnimatedValue()获取变化的值，并将这个值传递给 Toolbar 的布局参数 LayoutParams，设置高度 LayoutParams 的 height。最后，调用 ValueAnimator 的 start()方法，启动动画。同时，当 ToolBar 动画结束时，通过 AnimatorListenerAdapter 的 onAnimationEnd()接口回调，填充列表（RecyclerView）的数据和启动 Fab（FloatingActionButton）控件的动画，Fab 的动画是由隐藏尺寸（scale 是 0）转变为完整尺寸（scale 是 1）。

```
    private void collapseToolbar(int height) {
        TypedValue tv = new TypedValue();
        getTheme().resolveAttribute(android.R.attr.actionBarSize, tv, true);
        int   toolBarHeight   =   TypedValue.complexToDimensionPixelSize(tv.data,
getResources().getDisplayMetrics());
        // 动画, height->toolBarHeight, 468->168
        ValueAnimator valueAnimator = ValueAnimator.ofInt(height, toolBarHeight);
        valueAnimator.addUpdateListener(animation -> {
            ViewGroup.LayoutParams lp = mTToolbar.getLayoutParams();
            lp.height = (Integer) animation.getAnimatedValue();
            mTToolbar.setLayoutParams(lp);
        });
        valueAnimator.start();

        valueAnimator.addListener(new AnimatorListenerAdapter() {
            @Override public void onAnimationEnd(Animator animation) {
                super.onAnimationEnd(animation);
                mPhRecyclerAdapter.setItems(ModelItem.getFakeItems());
                ViewCompat.animate(mFabBar).setStartDelay(500)
                        .setDuration(500).scaleX(1).scaleY(1).start();
            }
        });
    }
```

列表项的适配器是一个标准的 RecyclerView 适配器，实现 onCreateViewHolder()、onBindViewHolder()、getItemCount()三个接口，onCreateViewHolder()用于填充布局，并将布局

关联至 ViewHolder；onBindViewHolder()用于绑定（bindTo）ViewHolder 的数据，即将数据放入布局中；getItemCount()设置列表显示项的个数。ViewHolder 作为静态类也存放在 ViewHolder 中，一个 ImageView 用于显示图片，一个 TextView 用于显示标题，通过 ButterKnife 绑定布局 ID，通过 Picasso 填充图片资源。setItems()是外部设置数据的接口，其核心是 notifyItemRangeInserted()方法，将数据列表顺次地加入到布局列表中，显示每个项目的动画效果，使得动画具有层次感。

```
    public static class PhRecyclerAdapter extends RecyclerView.Adapter<
PhRecyclerAdapter.PhViewHolder> {
        private final ArrayList<ModelItem> mItems = new ArrayList<>(); // 数据

        public void setItems(List<ModelItem> items) {
            // 启动动画的关键位，顺次添加动画效果
            mItems.addAll(items);
            notifyItemRangeInserted(0, mItems.size());
        }

        @Override public PhViewHolder onCreateViewHolder(ViewGroup parent, int
viewType) {
            View                                v                                =
LayoutInflater.from(parent.getContext()).inflate(R.layout.item_card,      parent,
false);
            return new PhViewHolder(v);
        }

        @Override public void onBindViewHolder(PhViewHolder holder, int position) {
            holder.bindTo(mItems.get(position));
        }

        @Override public int getItemCount() {
            return mItems.size();
        }

        public static class PhViewHolder extends RecyclerView.ViewHolder {
            @Bind(R.id.item_tv_title) TextView mTvTitle;
            @Bind(R.id.item_iv_image) ImageView mIvImage;
            private Context mContext;
            public PhViewHolder(View itemView) {
                super(itemView);
                ButterKnife.bind(this, itemView);
                mContext = itemView.getContext().getApplicationContext();
            }
            public void bindTo(ModelItem item) {
                Picasso.with(mContext).load(item.getImgId()).into(mIvImage);
                mTvTitle.setText(item.getName());
            }
```

```
        }
    }
```

具体的动画效果位于 ItemAnimatorFactory 类的静态方法 slidein()中，使用减速差值器（DecelerateInterpolator）创建 SlideInUpDelayedAnimator 动画，动画间隔 600 毫秒，SlideInUpDelayedAnimator 最终继承于 ItemAnimator，用于显示 RecyclerView 列表每一项的动画效果。

```
    public class ItemAnimatorFactory {
        static public RecyclerView.ItemAnimator slidein() {
            SlideInUpDelayedAnimator animator = new SlideInUpDelayedAnimator(new
DecelerateInterpolator(1.2f));
            animator.setAddDuration(600);
            return animator;
        }
    }
```

最有趣的是 SlideInUpDelayedAnimator 的实现，包含两个方法：preAnimateAdd()用于设置启动动画之前的状态，onAnimatedAdd 用于添加动画。在 preAnimateAdd()中，设置 ViewHolder 布局 itemView 的竖直偏移（TranslationY），偏移距离正好是 itemView 的高度，即起始位置位于下一个 itemView 的位置，这样才能执行向上偏移动画，同时设置 itemView 的透明度（Alpha）为全透明（值为 0）。在 onAnimatedAdd()中，创建视图属性动画（ViewPropertyAnimatorCompat），执行控件是 holder.itemView，偏移终点是 0，即最初位置，差值器是参数传递的差值器，启动时间是每一项间隔 offsetDelay 的毫秒数（200 毫秒）。

```
    public class SlideInUpDelayedAnimator extends BaseItemAnimator {
        private final int offsetDelay = 200;
        private final Interpolator mInterpolator;

        public SlideInUpDelayedAnimator(Interpolator interpolator) {
            mInterpolator = interpolator;
        }

        @Override
        protected void preAnimateAdd(RecyclerView.ViewHolder holder) {
            ViewCompat.setTranslationY(holder.itemView,
holder.itemView.getHeight());
            ViewCompat.setAlpha(holder.itemView, 0);
        }

        @Override
        protected ViewPropertyAnimatorCompat onAnimatedAdd(RecyclerView.ViewHolder
holder) {
            return ViewCompat.animate(holder.itemView)
                    .translationY(0)
                    .setInterpolator(mInterpolator)
```

```
                .setStartDelay(offsetDelay * holder.getLayoutPosition());
        }
    }
```

关于列表项动画的基类 BaseItemAnimator，可以参考源码。

最终，预留位置动画的页面如图 6-3 所示。

图 6-3　预留位置动画

本节主要讲解两种常见的动画组合效果，即效果显示动画和预留位置动画，避免页面的冷启动，提升用户体验。

（1）效果显示动画，以控件元素的变化为主，通过位置或形状上的改变，突出页面的核心元素，引导用户的关注或点击。

（2）预留位置动画，以控件元素的归位为主，通过其他位置转换为目标位置，在启动页面时，赋予控件更鲜活的个性。

在本例的讲解中，通过效果显示动画和预留位置动画，我们熟悉 TextView、ImageView、Button、ToolBar、RecyclerView、FloatingActionButton 等常用控件的动画形式，这些动画效果，如位移、透明度、尺寸等，也可以应用于其他控件，控件与动画可以灵活组合。同时，显示动

画和预留位置也并不互斥，在设计动画效果中，可以根据需求同时添加。需要注意的是，动画效果非常吸引用户的注意力，在页面中一定避免添加过多的动画效果，导致用户的注意力分散。要避免页面元素混乱，造成主次不分，合适的动画是突出想要突出的部分，而不是全盘重点，不要“手里握个锤子，看什么都是钉子”。一个好的产品，动画一定是必不可少，也是高级程序员必备的工具箱，希望大家熟练使用。

6.2　实现页面展开中圆形爆炸的动画效果

在 Android 系统的 5.0 版本中，引入 Material Design（材料设计）的设计理念，为应用带来大量绚丽的动画效果。随着厂商的版本迭代，约超过 1/2 的手机都已经是 5.0 版本以上的操作系统，如图 6-4 所示。随着更多低价、高性能低端手机的普及，Material Design 的动画效果会为选择 Android 系统的用户带来更好的体验，也会为 Android 应用大大地增加留存率。

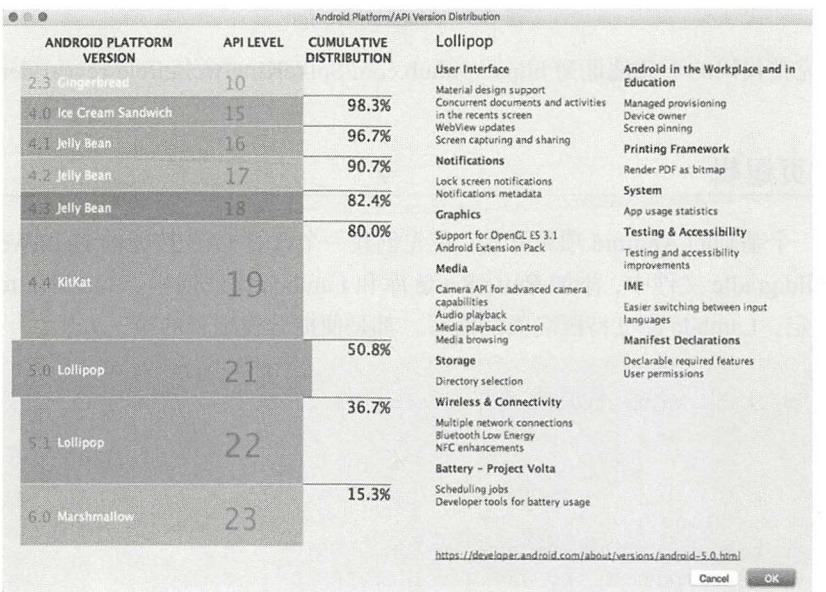


图 6-4　Android 系统的比例

在切换页面的过程中，最常用的效果之一就是 SharedElementTransition（分享元素转换），通过关联控件与页面，设置变换方式。当由控件进入页面时，把控件动态地转换为页面，当页面退出至控件时，把页面动态地转换为控件，同时，也支持设置控件的滑动轨迹。这样的控件样式非常适合展示消息通知，或者展示广告页面，为用户提供良好的体验。

在控件与页面的切换过程中，引入 CircularReveal（圆形展示）的动画效果。当控件转换为页面时，动画效果从控件的中心逐渐展开，直至填充为全部页面；当页面转换为控件时，动画效果从页面的边缘逐渐集中，直至聚合为控件形状。效果非常类似于爆炸和凝聚，如图 6-5 所示。

图 6-5　Material Design

本实例完整代码的下载地址为 https://github.com/SpikeKing/wcl-circle-reveal-demo。

6.2.1　首页逻辑

本例从一个基础的 Android 项目开始，首先创建一个含有 FAB 按钮的 HelloWorld 工程，在工程配置 build.gradle 文件中，添加 ButterKnife 库和 Lambda 库的依赖。ButterKnife 库支持布局 ID 的快速绑定，Lambda 库支持匿名类的简写，都是便捷开发项目的第三方库。

```
plugins {
    id "me.tatarka.retrolambda" version "3.2.5"
}

android {
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
}

dependencies {
    compile 'com.jakewharton:butterknife:7.0.1'
}
```

在 MainActivity 的入口函数 onCreate()中，调用 ButterKnife 的 bind()方法，将布局绑定当前页面。接着，使用 ButterKnife 的@Bind 注解，将布局 ID 的 fab 绑定至实例 FloatingActionButton

的 mFab。

```
    @Bind(R.id.fab) FloatingActionButton mFab;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ButterKnife.bind(this);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
    }
```

在 startOtherActivity()方法中，设置 FAB 按钮的跳转事件，根据 Android 系统的版本不同，选择不同的跳转样式。当 SDK 版本大于 5.0 时，即 SDK_INT 大于 LOLLIPOP，使用页面的场景转换动画（Scene Transition Animation）；对于其他情况，使用页面的默认跳转。在 SDK 5.0 及以上版本时，支持 Activity 选项，即 ActivityOptions，通过在选项中设置不同的动画。当切换页面时，会呈现出不同的效果。场景转换动画需要三个参数，即当前页面类（Activity）、跳转视图类（mFab）、分享元素名称（mFab.getTransitionName()）。当页面跳转至新的页面时，被分享视图（如 FAB 视图）会以动画形式转换为新的页面，在新的页面中，也含有一个视图，与被分享视图的分享元素名称相同，表示两个视图的连接状态。

```
    public void startOtherActivity(View view) {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
            ActivityOptions options =
                    ActivityOptions.makeSceneTransitionAnimation(this, mFab, mFab.
getTransitionName());
            startActivity(new Intent(this, OtherActivity.class), options.toBundle
());
        } else {
            startActivity(new Intent(this, OtherActivity.class));
        }
    }
```

在布局文件 activity_main.xml 中，FloatingActionButton（FAB）是本例重要的视图组件。在 onClick 属性中，添加点击事件逻辑 startOtherActivity 方法。在 transitionName 属性中，添加转换名称 other_transition_name，需要与新的页面中相应视图的转换名称相同。同时，设置 tools:targetApi 属性为 lollipop（中文是棒棒糖□），表示页面的显示样式是 Android 5.0 版本。

```
    <android.support.design.widget.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|end"
        android:layout_margin="@dimen/fab_margin"
        android:clickable="true"
```

```
    android:onClick="startOtherActivity"
    android:src="@android:drawable/ic_dialog_email"
    android:transitionName="@string/other_transition_name"
    tools:targetApi="lollipop"/>
```

首页的效果，关注右下角的 FAB 按钮，如图 6-6 所示。

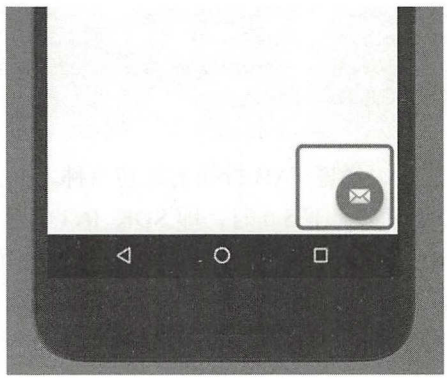


图 6-6　FAB 按钮

6.2.2　新页逻辑

在新页布局 activity_other.xml 中，含有三个部分：变换控件 FloatingActionButton（FAB）、背景控件 TextView、关闭控件 ImageView。

◎　变换控件 FAB：新页的变换控件 FAB 与首页的变换控件 FAB 一一对应，拥有相同的转换名称（transitionName）属性；layout_centerInParent 表示位置居中；backgroundTint 表示背景着色；elevation 表示控件悬空高度；fabSize 表示 FAB 控件大小；pressedTranslationZ 表示按压时竖直（Z 轴）变化。

◎　背景控件 TextView：尺寸 layout_width 和 layout_height 表示控件铺满整个屏幕；background 表示背景色；gravity 表示位置居中；text 表示控件中文本；visibility 表示隐藏控件，如果需要观察控件的样式，将开发工具 tools 的 visibility 属性设置为 true。

◎　关闭控件 ImageView：onClick 表示点击逻辑，位于 backActivity 方法中；src 表示控件中的图片，白色叉型图片；visibility 的设置方式与背景控件相同，都是隐藏控件，在开发工具中显示。

```
<RelativeLayout
    android:id="@+id/other_rl_container"
    android:layout_width="match_parent"
```

```
    android:layout_height="match_parent"
    android:background="@android:color/transparent">

    <android.support.design.widget.FloatingActionButton
        android:id="@+id/other_fab_circle"
        android:layout_centerInParent="true"
        android:transitionName="@string/other_transition_name"
        app:backgroundTint="@color/colorAccent"
        app:elevation="0dp"
        app:fabSize="normal"
        app:pressedTranslationZ="8dp"
        tools:targetApi="21"/>

    <TextView
        android:id="@+id/other_tv_container"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="@color/colorAccent"
        android:gravity="center"
        android:text="@string/other_text"
        android:visibility="invisible"
        tools:visibility="visible"/>

    <ImageView
        android:id="@+id/other_iv_close"
        android:clickable="true"
        android:onClick="backActivity"
        android:src="@drawable/ic_close_white"
        android:visibility="invisible"
        tools:visibility="visible"/>
</RelativeLayout>
```

在新页 OtherActivity 的入口函数 onCreate()中，同样使用 ButterKnife 绑定页面。当 Android SDK 的版本大于或等于 5.0 时，在创建（入场）和退出（退场）页面中使用动画效果，即 setupEnterAnimation()和 setupExitAnimation()。创建是爆炸效果，退出是凝聚效果。对于其他情况，默认初始化视图，提供简易的动画效果。

```
@Override protected void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_other);
    ButterKnife.bind(this);

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
        setupEnterAnimation(); // 入场动画
        setupExitAnimation(); // 退场动画
    } else {
        initViews();
    }
```

```
    }
```

本例的入场动画和退场动画会单独分析。在默认初始化视图 initViews()方法中，使用 Handler 创建异步执行方法，方法位于主线程 Looper.getMainLooper()中。在异步执行方法中，创建延迟的淡入（fade_in）动画 Animation，动画延迟 300 毫秒（setDuration(300)），将背景控件和关闭控件设置为可视，使用淡入动画逐渐显示。从而在创建页面中，实现一个简易的动画效果，这也是在 Material Design 动画方式之前常用的动画形式。

```
    private void initViews() {
        new Handler(Looper.getMainLooper()).post(() -> {
            Animation animation = AnimationUtils.loadAnimation(this,
android.R.anim.fade_in);
            animation.setDuration(300);

            mTvContainer.startAnimation(animation);
            mIvClose.setAnimation(animation);
            mTvContainer.setVisibility(View.VISIBLE);
            mIvClose.setVisibility(View.VISIBLE);
        });
    }
```

当点击关闭控件时，调用 backActivity()方法，执行关闭逻辑。当 Android SDK 的版本大于或等于 5.0 时，执行动画版本覆写的关闭逻辑 onBackPressed()，否则执行默认的关闭逻辑 defaultBackPressed()。

```
    public void backActivity(View view) {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
            onBackPressed();
        } else {
            defaultBackPressed();
        }
    }

    private void defaultBackPressed() {
        super.onBackPressed();
    }
```

新页的效果，如图 6-7 所示。

图 6-7　新页的效果

6.2.3　显示动画

在入场动画 setupEnterAnimation()中，首先使用转换填充器 TransitionInflater 创建转换动画 Transition，动画属性位于 R.transition.arc_motion 中。接着，在转换动画 Transition 中添加转换监听，当转换动画完成时，回调接口 onTransitionEnd()。在接口中，调用 Transition 的 removeListener()方法删除当前监听，同时调用 animateRevealShow()方法执行爆炸动画，展示全部页面。最后，调用 setSharedElementEnterTransition()设置页面窗口（getWindow()）的分享元素入场转换属性。注解@TargetApi 表示 setupEnterAnimation()方法只能应用于 SDK 5.0 版本以上，如果错误使用，在编译阶段时，IDE 会发出警告。这一部分效果是当点击首页右下角的 FAB 按钮时，FAB 按钮从右下角向下旋转 90° 至页面中心。

```
@TargetApi(Build.VERSION_CODES.LOLLIPOP)
private void setupEnterAnimation() {
    Transition transition = TransitionInflater.from(this)
            .inflateTransition(R.transition.arc_motion);
    transition.addListener(new Transition.TransitionListener() {
        @Override public void onTransitionStart(Transition transition) {}
        @Override public void onTransitionEnd(Transition transition) {
            transition.removeListener(this);
```

```
            animateRevealShow();
        }
        @Override public void onTransitionCancel(Transition transition) {}
        @Override public void onTransitionPause(Transition transition) {}
        @Override public void onTransitionResume(Transition transition) {}
    });
    getWindow().setSharedElementEnterTransition(transition);
}
```

旋转转换的动画效果位于 arc_motion 文件中，差值器 interpolator 的 linear_out_slow_in 属性表示线性滑动、缓慢滑入；间隔 duration 设置为 500 毫秒；滑动轨迹 changeBounds 设置角度动画 arcMotion，最大角度为 90° ，最小水平角度为 90° ，最小竖直角度为 0°，效果为在控件的滑动过程中，向下旋转 90°。

```
<transitionSet
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:duration="500"
    android:interpolator="@android:interpolator/linear_out_slow_in">

    <changeBounds>
        <!--suppress AndroidElementNotAllowed -->
        <arcMotion
            android:maximumAngle="90"
            android:minimumHorizontalAngle="90"
            android:minimumVerticalAngle="0"/>
    </changeBounds>
</transitionSet>
```

爆炸的动画效果位于 animateRevealShow()方法中，核心是调用 GuiUtils 的静态方法 animateRevealShow()，第一个参数是 Context，页面的进程信息；第二个参数是最终显示布局 mRlContainer；第三个参数是爆炸起始半径，即 mFabCircle.getWidth()/2；第四个参数是爆炸展开的颜色，即 R.color.colorAccent；最后一个参数是动画状态接口，onRevealHide()是关闭状态的回调，onRevealShow()是展示状态的回调，爆炸动画的效果最终是展开，因此覆写 onRevealShow()方法，调用 initViews()初始化视图。注解@TargetApi 同样是用于版本状态检查。

```
@TargetApi(Build.VERSION_CODES.LOLLIPOP)
private void animateRevealShow() {
    GuiUtils.animateRevealShow(
            this, mRlContainer,
            mFabCircle.getWidth() / 2, R.color.colorAccent,
            new GuiUtils.OnRevealAnimationListener() {
                @Override public void onRevealHide() {}
                @Override public void onRevealShow() {
                    initViews();
                }
            });
```

```
    }
```

爆炸的动画效果核心是 GuiUtils 的 animateRevealShow()方法。首先，获取填充视图 view 的中心，即 cx 和 cy，也是变换的中心坐标；其次，获取变换圆圈的最终半径，即水平和竖直距离的斜边长。接着，调用 createCircularReveal()创建圆形展示动画（Circular Reveal），参数是填充视图 view、中心坐标 cx 和 cy、起始半径 startRadius、终止半径 finalRadius。进而设置动画 Animator 属性，即持续时间 300 毫秒、先加速后减速差值器、动画监听适配器。在监听适配器中，处理动画启动 onAnimationStart()和动画结束 onAnimationEnd()两种状态，在动画启动方法中，设置填充视图 view 的背景颜色为指定参数颜色 color，在动画结束方法中，设置填充视图 view 为可视状态，同时回调 OnRevealAnimationListener 的 onRevealShow()方法，在外部继续执行其他逻辑。最后，调用 Animator 的 start()方法，启动动画效果。

```
    @TargetApi(Build.VERSION_CODES.LOLLIPOP)
    public static void animateRevealShow(
            final Context context, final View view,
            final int startRadius, @ColorRes int color,
            OnRevealAnimationListener listener) {
        int cx = (view.getLeft() + view.getRight()) / 2;
        int cy = (view.getTop() + view.getBottom()) / 2;

        float finalRadius = (float) Math.hypot(view.getWidth(), view.getHeight());

        // 设置圆形显示动画
        Animator anim = ViewAnimationUtils.createCircularReveal(view, cx, cy,
startRadius, finalRadius);
        anim.setDuration(300);
        anim.setInterpolator(new AccelerateDecelerateInterpolator());
        anim.addListener(new AnimatorListenerAdapter() {
            @Override public void onAnimationEnd(Animator animation) {
                super.onAnimationEnd(animation);
                view.setVisibility(View.VISIBLE);
                listener.onRevealShow();
            }

            @Override public void onAnimationStart(Animator animation) {
                super.onAnimationStart(animation);
                view.setBackgroundColor(ContextCompat.getColor(context, color));
            }
        });

        anim.start();
    }
```

6.2.4 退出动画

退出动画分为两个部分，一个是 setupExitAnimation()方法，一个是 onBackPressed()方法。

在退场动画 setupExitAnimation()中，首先创建淡出类 Fade 类，持续时间为 300 毫秒；接着，调用页面窗口（getWindow()）的 setReturnTransition()方法，设置退出动画，让页面以淡出的样式逐渐消失。设置入场动画的接口是 setSharedElementEnterTransition()，设置出场动画的接口是 setReturnTransition()。

```
@TargetApi(Build.VERSION_CODES.LOLLIPOP)
private void setupExitAnimation() {
    Fade fade = new Fade();
    fade.setDuration(300);
    getWindow().setReturnTransition(fade);
}
```

另一个退场动画是凝聚效果，当点击右上角的白色叉号或返回键时，都会触发 onBackPressed()接口，执行退场动画。退场动画的核心逻辑位于 GuiUtils 的静态方法 animateRevealHide()中，参数是页面的 Context、凝聚动画的填充视图 mRlContainer、凝聚圆形的最终填充颜色 colorAccent、凝聚状态的回调接口 OnRevealAnimationListener。在回调接口中，监听展示状态 onRevealShow()和隐藏状态 onRevealHide()，退出动画最终是隐藏，因此覆写隐藏接口，继续执行默认的退出操作。

```
@Override public void onBackPressed() {
    GuiUtils.animateRevealHide(
            this, mRlContainer,
            mFabCircle.getWidth() / 2, R.color.colorAccent,
            new GuiUtils.OnRevealAnimationListener() {
                @Override
                public void onRevealHide() {
                    defaultBackPressed();
                }

                @Override
                public void onRevealShow() {

                }
            });
}
```

退出操作调用父类 AppCompatActivity 的退出接口 onBackPressed()，即默认的退出页面。

```
private void defaultBackPressed() {
    super.onBackPressed();
}
```

凝聚的动画效果核心是 GuiUtils 的 animateRevealHide()方法，与爆炸效果类似，不同的点是起始半径和终止半径正好相反，爆炸是由小到大，凝聚是由大到小。同时，动画状态的监听接口也不同，当动画起始 onAnimationStart()时，设置填充视图 view 的背景颜色；当动画关闭 onAnimationEnd()时，先回调外层的隐藏接口 onRevealHide()，再设置填充视图 view 为不可视。

```
    @TargetApi(Build.VERSION_CODES.LOLLIPOP)
    public static void animateRevealHide(
            final Context context, final View view,
            final int finalRadius, @ColorRes int color,
            OnRevealAnimationListener listener
    ) {
        int cx = (view.getLeft() + view.getRight()) / 2;
        int cy = (view.getTop() + view.getBottom()) / 2;
        int initialRadius = view.getWidth();
        // 与入场动画的区别是圆圈起始和终止的半径相反
        Animator anim = ViewAnimationUtils.createCircularReveal(view, cx, cy,
initialRadius, finalRadius);
        anim.setDuration(300);
        anim.setInterpolator(new AccelerateDecelerateInterpolator());
        anim.addListener(new AnimatorListenerAdapter() {
            @Override public void onAnimationStart(Animator animation) {
                super.onAnimationStart(animation);
                view.setBackgroundColor(ContextCompat.getColor(context, color));
            }

            @Override public void onAnimationEnd(Animator animation) {
                super.onAnimationEnd(animation);
                listener.onRevealHide();
                view.setVisibility(View.INVISIBLE);
            }
        });
        anim.start();
    }
```

好的动画可以为应用带来震撼的效果，提升产品的需求质量，抓住用户的核心注意力，让用户觉得更有趣，更愿意使用应用。本例的动画效果分为入场动画和退场动画，通过分享元素（Shared Element）FAB 控件，将控件与页面连接到一起。在入场时，控件转换为页面；在退场时，页面转换为控件。分享元素的本质是在不同的页面中创建相同的元素，两者通过动画相互关联。本例在分享元素的基础之上，或是将分享元素通过爆炸动画成为最终的页面，或是将页面通过凝聚成为最终的分享元素，达到更优雅的动画效果。在开发动画效果的过程中，不要拘泥于一种样式，而是要组合多种已有的动画效果，最终实现设计的需求。美好且精致的产品是高级程序员的最终追求之一，大道至极。

第 7 章
Kotlin 与 SVG

7.1 Kotlin 基础教程

Kotlin 是另一种编程语言，基于 JVM（Java Virtual Machine，Java 虚拟机），由 JetBrains 公司开发，兼容 Java 编程语言。IntelliJ IDEA、PyCharm、Android Studio 等 IDE（Integrated Development Environment，集成开发环境），都是出自 JetBrains 公司，因此 JetBrains 拥有丰富的编程语言架构经验。Kotlin 的经典介绍：

Statically typed programming language for the JVM, Android and the browser. 100% interoperable with Java™.

适用于 JVM、Android、浏览器的静态类型编程语言，完全（100%）兼容于 Java。

在 Kotlin 的介绍中，已经充分强调支持 Android 项目的开发。由于 Kotlin 语言的发布时间要晚于 Java 语言，在 Kotlin 语言中，加入很多新的、实用的语言特性，在开发的过程中，更加便捷，容错性也更好，类似于 Python 编程语言，体验非常好。因此，Kotlin 也作为 Android 社区中比较活跃的热点，逐渐走向成熟，其稳定版本已经发布，同时也得到了一些著名 Android 开发者的关注，毕竟 Android Studio 是基于 JetBrains 的框架，同宗同源，Java 文件可以直接转换为 Kotlin 文件，两者具有非常强的互操作性。

Kotlin 语言非常精练、简洁，对于 Android 开发也是非常友好，是一种非常优雅的开发语言。相比之下，早期的 Cpp、中期的 Java 至目前的 Python，每一种语言都有各自的优势。但总的来讲，如果不考虑性能，越新的语言越容易实现基础的功能。当年用 Cpp 语言独立开发大型

的人工智能算法项目，每一段逻辑都需要编写大量的代码，处理类与类的解耦、管理实例的内存、建立合适的继承体系，都需要开发者独立应对，着实麻烦；而使用 Python 语言实现类似的功能就非常简洁，性能在一些优化的函数库中损耗也很小。作为一个开发者，永远都要拥抱未来，真正的技术都来源于一行一行的代码，而编程语言就是我们的思考方式，学习换一种思维，总会是一件有趣的事情。

本文分为两个部分，讲解 Kotlin 最精髓的部分：

◎　基础部分，初探 Kotlin 与集成 Android 项目。

◎　进阶部分，优雅地扩展 Kotlin 类的方法和属性。

通过本例的学习，掌握Kotlin的全部开发技巧，来看看如何在已有Android项目中集成Kotlin吧！

Talk is cheap, show you the code.

本文实例完整代码的下载地址为 https://github.com/SpikeKing/wcl-kotlin-demo。

7.1.1　基础部分

基础部分主要有两部分，即：

◎　基于 Kotlin 的 Android 工程的配置与构建。

◎　把已有的 Android 工程转换为 Kotlin 工程。

Kotlin 编程语言也在不断地迭代与更新，最新的配置方式可能与本文介绍的有所不同，但是核心框架是不变的，也会向下兼容，不妨碍读者学习 Kotlin 框架的精髓。

1. 工程配置

本例从一个基础的 HelloWorld 版的 Android 工程开始。在整个工程配置的 build.gradle（最外层）中，添加 Kotlin 的版本号，为了构建脚本与依赖库的版本统一，其余不做修改。

```
buildscript {
    ext.kotlin_version = "1.0.4"
    // ...
}
```

接着，在项目的 build.gradle（app 文件夹）中，添加 Kotlin 的构建脚本和依赖库，版本号是外层的 Gradle 配置中的版本号。JetBrains 组织名称是 org.jetbrains.kotlin，构建脚本库是 kotlin-gradle-plugin，依赖库是 kotlin-stdlib。

```
buildscript {
    // ...
```

```
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}
// ...
dependencies {
    // ...
    compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
}
```

添加 Kotlin 支持 Android 的插件：kotlin-android 和 kotlin-android-extensions，这些插件就在构建脚本的 kotlin-gradle-plugin 库中被定义。kotlin-android 用于编译 Android 代码；kotlin-android-extensions 用于扩展布局的绑定关系。

```
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'
```

添加 Java 源码的编译路径。在 main 中创建 kotlin 文件夹，用于存放 Kotlin 的开发源码，同时，在 Android 的 Gradle 架构中声明源码目录的位置。

```
android {
    // ...
    sourceSets {
        main.java.srcDirs += 'src/main/kotlin'
    }
}
```

完整的支持 Kotlin 的模块 build.gradle.

2. 插件安装

同时，需要安装 Android Studio 的 Kotlin 插件，用于支持 Kotlin 的语言特性。

选择：【Preferences（偏好）】 -> 【Plugins（插件）】，搜索 Kotlin 插件，安装即可，如图 7-1 所示。

只需要安装 Kotlin 插件即可，而 Kotlin Extensions 插件已经集成于 Kotlin 插件中。

在安装完成插件之后，需要重新启动 Android Studio，完成 IDE 中的插件加载，支持 Kotlin 插件的功能，将一个支持 Java 编程语言的 Android Studio，转换为同时支持 Kotlin 编程语言。

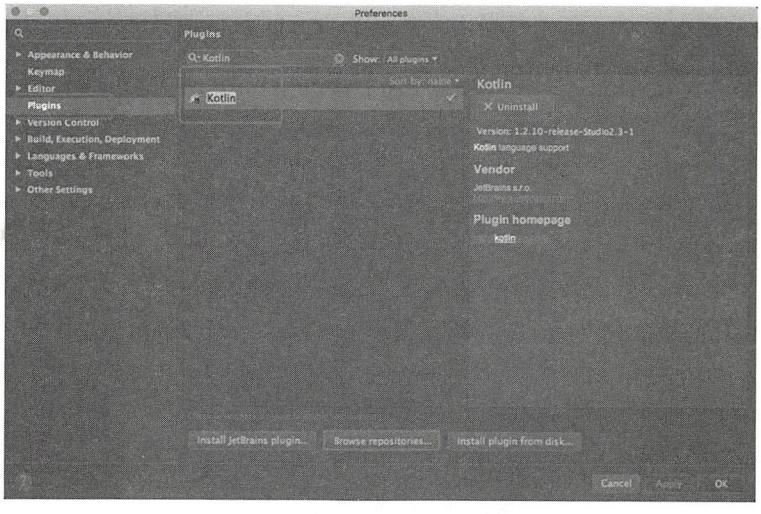


图 7-1　Kotlin 插件

3. 文件转换

在 Android Studio 安装 Kotlin 插件之后，就支持将 Java 源码无损的转换为 Kotlin 源码。

第一种方法，选中待转换的 Java 文件，如 MainActivity.java，使用“Command+Shift+A”命令启动活动（Action）搜索框，在搜索框中输入“Convert”，找到转换功能，点击即可转换，如图 7-2 所示。

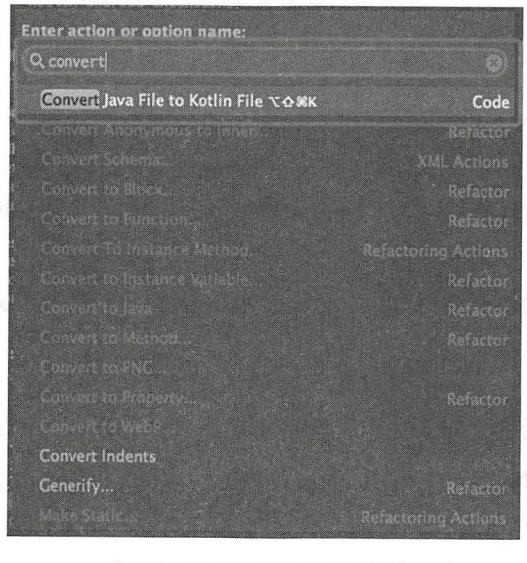


图 7-2　Kotlin 转换方法（一）

第二种方法，在 IDE 的顶部工具栏中，先选择“Code”，再选择“Convert Java File to Kotlin File”，也可以将 Java 源码转换为 Kotlin 源码，如图 7-3 所示。

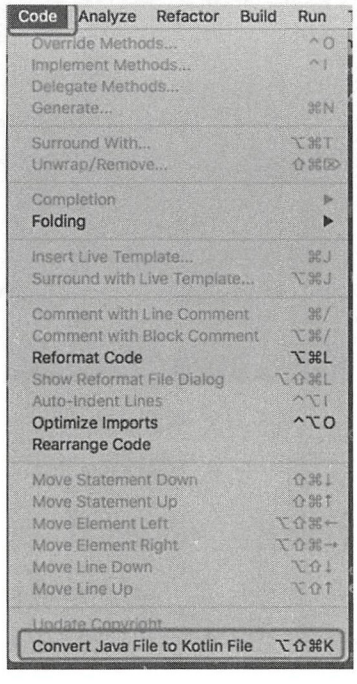


图 7-3　Kotlin 转换方法（二）

第三种方法，就是记住快捷键，“Alt + Shift + Command + K”，多个按键的快捷键组合，也可以实现转换功能。

Java 源码的后缀以“.java”结尾，而 Kotlin 源码的后缀以“.kt”结尾。将转换后的文件，如 MainActivity.kt，由 Java 的源码文件夹（src/main/java）剪切至 Kotlin 的源码文件夹（src/main/kotlin）中即可使用。推荐将 Kotlin 文件和 Java 文件分别存放至不同的目录，不过两者放在一起也可以使用。

这样就可以将已有的 Java 版本的 Android 工程，无缝地转换为 Kotlin 版本。

4. 源码分析

接着，将分析转换后的 Kotlin 源码，就是非常经典的 MainActivity.kt。

◎　MainActivity 类继承于 AppCompatActivity 类；class 是类的关键字。

◎　重写 onCreate()方法；override 是重写关键字；fun 是函数关键字；“?”支持参数为空，即注解@Nullable；

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        // ...
    }
}
```

同时，在 Kotlin 中，设置布局属性也非常方便，控件 TextView 的 ID 是 main_tv_message，支持通过给属性赋值的方式设置属性值，如文本内容 text 属性和字体尺寸 textSize 属性，不需要任何前置声明，比 Java 简洁明快。

```
main_tv_message.text = "Hello Kotlin"
main_tv_message.textSize = 20.0f
```

使用属性 ID 的这些功能，需要导入对应的 Kotlin 布局相关的包 kotlinx.android.synthetic，指定布局文件名称 activity_main 。

```
import kotlinx.android.synthetic.main.activity_main.*
```

这样，数十行的 Java 源码，仅仅需要 10 行 Kotlin 源码就能实现相同的功能，简单而优雅，这就是语言的魅力。

```
import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import kotlinx.android.synthetic.main.activity_main.*

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        main_tv_message.text = "Hello Kotlin"
        main_tv_message.textSize = 20.0f
    }
}
```

最终的页面效果如图 7-4 所示。

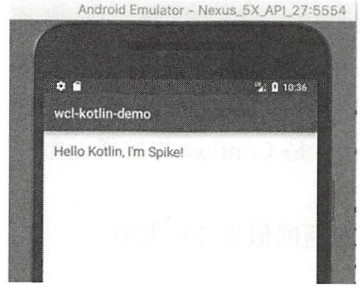


图 7-4　Hello Kotlin

至此，基础部分已经完成，创建了一个 Kotlin 的 Hello World 工程，实现一些简易的功能。程序员有一句古语："会写一个语言的 Hello World，就学会了这个语言一半！"当然，这是一句俏皮话，每个语言都有丰富的特性，还需要在开发的过程中逐渐学习，本文只为大家打开了一扇编程语言的门，门里的宝藏还需要读者凭着努力一点一点地获取。

7.1.2 进阶部分

除了创建基础的 Kotlin 工程，接着介绍 Kotlin 一些有趣的特性，起到抛砖引玉的作用，增加大家学习语言的乐趣。本例讲述三个语言特性，即类型安全的构建模式（Type-Safe Builder Pattern）、扩展函数（Extension Function）、扩展属性（Extension Property）。

1. 类型安全的构建模式

类型安全的构建模式类似模板的功能，提供模板参数，用于构建任意类型的视图或视图组的实例，比较适用于构建嵌套式数据结构，比如 XML 格式的布局文件。不过，在实际开发中，视图或视图组尽量还要通过文件的形式设置，避免在类中编写。本例构建一个方法 v，用于创建视图 View 模板，源码位置位于 MainActivity.kt 中。

```
inline fun <reified TV : View> Context.v(init: TV.() -> Unit): TV {
    val cns = TV::class.java.getConstructor(Context::class.java)
    val view = cns.newInstance(this)
    view.init()
    return view
}
```

（1）声明部分

◎ inline fun：inline 是内联关键字，表明 v()方法是内联方法，支持泛型功能。

◎ <reified TV : View>：reify 的含义是"具体化"，作为 Kotlin 方法的泛型关键字，表示在方法体中，支持访问指定泛型的类对象，如 View，必须以内联方式（inline）声明这个方法，泛型才有效。这段代码的意思就是 v()方法，使用命名为 TV（即 Type of View，视图类型)的 reified 泛型，TV 必须是 View 或者 View 的子类。

◎ Context.v(init: TV.() -> Unit)：init 是 v()方法的参数；init 是 TV 类型的 Lambda 方法；调用 TV 的构造器（TV.()）创建 TV 的实例，返回 Unit（类似于 Java 的 Void）；Context.v()表示 v 是 Context 的扩展函数，支持在 Context 或 Context 的子类中使用。

◎ ：TV：表示 v()函数的返回值是 TV 类型。

（2）逻辑部分

◎　调用 TV 类对象的 getConstructor()方法，获取构造器 cns，参数是 Context 类对象。

◎　调用构造器 cns 的 newInstance()方法，获取 TV 对象的实例 view。

◎　调用实例的 init()方法，初始化实例 view。

◎　返回 TV 类对象的已初始化实例 view。

其中，TV::class.java 和 Context::class.java 表示 TV 泛型的类对象和 Context 的类对象。

同样地，构建一个方法 v，用于创建视图组 ViewGroup 模板，与视图方法类似。不同的是，在视图组的函数内部，增加一个添加视图的 addView()方法，将泛型参数 V 的实例 view 添加至视图组中。

```
inline fun <reified V : View> ViewGroup.v(init: V.() -> Unit): V {
    val cns = V::class.java.getConstructor(Context::class.java)
    val view = cns.newInstance(context)
    addView(view)
    view.init()
    return view
}
```

使用视图 View 模板创建视图组 LinearLayout，设置布局参数 layoutParams，设置布局方向 orientation，直接给属性赋值即可，非常简洁。

```
val view = v<LinearLayout> {
    layoutParams = LayoutParams(LayoutParams.WRAP_CONTENT, LayoutParams.
WRAP_CONTENT)
    orientation = LinearLayout.VERTICAL
    // ...
}
```

使用视图组 ViewGroup 模板创建 TextView，因为视图 TextView 是需要设置到外层视图组 LinearLayout 中，即调用视图组的 addView()方法，这个方法已经包含在视图组 ViewGroup 模板中。在 TextView 视图中，设置布局参数 layoutParams，设置文本内容 text，调用 TextView 的内部方法 setTextColor()设置颜色值。在视图模板 v 的内部，相当于是 TextView 的子类，可以调用 TextView 类内部的方法。

```
val view = v<LinearLayout> {
    layoutParams = LayoutParams(LayoutParams.WRAP_CONTENT, LayoutParams.
WRAP_CONTENT)
    orientation = LinearLayout.VERTICAL

    v<TextView> {
        layoutParams = LayoutParams(LayoutParams.WRAP_CONTENT, LayoutParams.
WRAP_CONTENT)
        text = "Hello"
```

```
            setTextColor(ContextCompat.getColor(applicationContext,
R.color.colorAccent))
        }

        v<TextView> {
            layoutParams = LayoutParams(LayoutParams.WRAP_CONTENT, LayoutParams.
WRAP_CONTENT)
            text = "World"
            setTextColor(ContextCompat.getColor(applicationContext,
R.color.colorAccent))
        }
    }
```

注意，Kotlin 还提供大量直接访问的类，如 applicationContext 等，非常便捷，省略无参数的 get()调用，类似于 Java 中的已赋值变量，同理，还有 resources、theme 等。

```
    Context applicationContext = getApplicationContext()  // 即 applicationContext
```

类型安全的构建模式（Type-Safe Builder Pattern）非常有趣，也很有用，省略了大量无异议的实例化方法，多个相同类型的子类可以复用同一个模式，除了在嵌套布局中，在其他方面也可以使用。

2. 扩展函数和属性

Kotlin 编程语言的另一个重要的特性是扩展函数和属性。扩展类的函数，即 Extension Function，可以在已有类中添加新的方法，比继承的形式更加便捷和优雅。

如在视图 View 中，添加 dp 转换函数 dp_f()，将 dp 值转换为像素值的浮点数；再添加 dp 转换函数 dp_i()，调用 dp_f()，将像素值的浮点数转换为像素值的整型。声明方式就是 View.xxx()，表示在 View 中扩展函数 xxx()，参数是浮点数 dp: Float，返回值也是浮点数 Float。在 View 中的 context 和 resources，都是 View 类中提供的直接访问变量。

```
    fun View.dp_f(dp: Float): Float {
        // 引用 View 的 context
        return TypedValue.applyDimension(
                TypedValue.COMPLEX_UNIT_DIP, dp, context.resources.displayMetrics)
    }

    fun View.dp_i(dp: Float): Int {  // 转换 Int
        return dp_f(dp).toInt()
    }
```

扩展类的属性，即 Extension Property，可以在已有的类中添加新的属性，使用等号“=”赋值，直接访问或使用。

如在视图 View 中，添加 padLeft 属性，提供 set()和 get()方法。在 set()方法中，将 paddingLeft

值替换为参数 value 值，其他值保持不变，如 paddingTop、paddingRight、paddingBottom，使用 View 类中的直接访问变量。在 get()方法中，直接返回已有的 paddingLeft，也是直接访问变量。注意，在扩展函数中，声明的是 fun，表示函数，而在扩展属性中，声明的是 var，表示变量，并且没有大括号“{}”。

```
    var View.padLeft: Int
        set(value) {
            setPadding(value, paddingTop, paddingRight, paddingBottom)
        }
        get() {
            return paddingLeft
        }
```

把 View 中的扩展函数 dp_i()和扩展属性 padLeft，直接在 TextView 的模板实例中应用，效果是把 Hello 文本向左填充 20dp。

```
    v<TextView> {
        layoutParams   =   LayoutParams(LayoutParams.WRAP_CONTENT,   LayoutParams.
WRAP_CONTENT)
        text = "Hello"
        setTextColor(ContextCompat.getColor(applicationContext,
R.color.colorAccent))
        padLeft = dp_i(20.0f);
    }
```

最终的效果如图 7-5 所示。

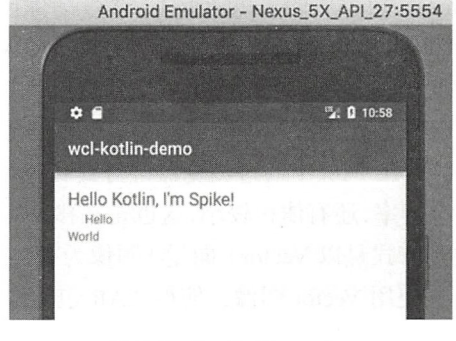


图 7-5　Kotlin Extension

Kotlin 语言与 Java 语言完全兼容，这无疑为工程的迁移打开了大门。Java 语言在本质上脱胎于 C/Cpp 语言，主要解决支持面向对象的问题，对于简化编程逻辑和注入对象功能方面比较欠缺，这是时代的局限性；而 Kotlin 语言的设计理念已经比较完善，优化了开发和容错逻辑，但是由于目前以 Java 语言为主的项目和函数库较多，Kotlin 语言替换 Java 语言仍需要一定的时间。

本例完整地剖析了 Kotlin 编程语言的配置方式，与已有 Android 项目的融合，重点讲解语言的两个重要的技术难点，即类型安全的构建模式、扩展函数和属性等。这些 Kotlin 语言的扩展技巧在编写简单的程序时可能没有多大的用处，但是当项目非常复杂时，灵活地扩展类可以大量地减少耦合。“麻雀虽小，五脏俱全。”教程带领大家理解 Kotlin 语言的基础概念，在开发时还需要继续不断地学习其他技巧，同时，技术较好的读者也可以直接阅读源码：

源码：https://github.com/JetBrains/kotlin

关于 Kotlin 语言的应用，这里涉及为项目如何选择合适的开发语言的问题，从经验上讲，这是一个复杂的问题。理论上来说，重要的项目仍需要保持原有的开发习惯，以最普遍的开发语言为主，如 Java 语言；而新的项目，如果技术团队中极客比较多，则可以选择以较新的开发语言为主，如 Kotlin 语言，但是在开发过程中可能会遇到技术难点，需要团队合力解决。如果使用较新的语言开发项目，一方面可以为团队提前积累技术经验，避免技术更新换代的手忙脚乱和无人可用，另一方面可以为探索新技术提供方法论，毕竟不同技术之间的思维方式都相通的。但是，对于初期技术团队，或者规模较小的团队，建议尽量不要使用新的技术，产品是最需要优先保证的，高级程序员一定要有产品意识，毕竟“Make Money”的事情总是最重要的事情。

7.2 SVG 基础教程

随着 Android 系统的更新换代，SDK 中也逐渐兼容了很多成熟的技术方案，比如 SVG 图像技术（Scalable Vector Graphics，可缩放的矢量图形）。SVG 图像技术是目前较为新颖的图像文件格式，基于 XML（Extensible Markup Language）框架，由 World Wide Web Consortium（W3C）联盟负责开发。从严格意义上来说，SVG 图像技术属于开放标准的矢量图形语言，设计者直接使用代码去描绘图像，在任何文字处理工具中都可开发，也支持通过改变部分代码，使图像具有交互功能，也支持插入至 HTML 网页中通过浏览器进行查看。同时，SVG 图像技术最重要的特性之一是支持无损地扩展分辨率，还有体积较小，这也是在移动应用中的两个优势。在 Android 系统中，SVG 图像技术的实现方式是以 Vector（向量）图像为主。在 Android 支持库 v23.2 中，AppCompat 的相关类已经开始使用 Vector 图像，使得 AAR 包的体积减少约 9%，约 70KB，惠及所有以此为基础的 Android 应用。

关于 SVG 图像技术的更多信息，可参考：https://zh.wikipedia.org/wiki/可缩放向量图形。

关于 SVG 图像技术在 Android 开发中的应用，本文主要分为两部分：

（1）基于 SVG 图像，创建 Vector 图像，同比于 PNG 图像等，可减少应用体积并支持无损放大。

（2）基于 SVG 编码，绘制 Vector 图像的路径动画，同比于 Gif 动画等，可减少应用体积并

支持无损放大；

本文实例完整代码的下载地址为 https://github.com/SpikeKing/TestSVG。

7.2.1　Vector 图像

SVG 图像技术的本质是通过编码将图像颜色块填充至图像上，类似于含有多个向量的图像。因此，在 Android 系统中，SVG 图像技术的实现方式是 Vector 图像。下面介绍如何生成和使用 Vector 图像。

1. 工程配置

首先，创建一个简单的 HelloWorld 工程，里面仅仅含有一个空白页面 MainActivity。在页面 MainActivity 中，添加一个跳转按钮 mBImage，跳转至用于显示 Vector 图像的页面 ImageActivity。

```
@Override protected void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    ButterKnife.bind(this);

    // Vector 图像
    mBImage.setOnClickListener(new View.OnClickListener() {
        @Override public void onClick(View v) {
            startActivity(new Intent(MainActivity.this, ImageActivity.class));
        }
    });

    // ...
}
```

在页面 ImageActivity 中，逻辑也非常简单，仅仅用于展示布局 activity_image，而 Vector 图像设置于布局 activity_image 中。

```
public class ImageActivity extends AppCompatActivity {
    @Override protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_image);
    }
}
```

在布局 activity_image 中，逻辑同样非常简单，仅仅用于展示 Vector 图像 v_homer_simpson，而 Vector 图像就是本节的核心。

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    android:orientation="vertical">

    <ImageView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:src="@drawable/v_homer_simpson"/>
</LinearLayout>
```

2. 创建图像

首先，选择 SVG 格式的图像，本例选择卡通人物辛普森（Simpson）的一张图片。在 Android Studio 中，依次选择：【File】 / 【New】 /【Vector Asset】，勾选“Local file （SVG，PSD）”复选框，并且选择 SVG 格式图像的加载路径，如图 7-6 所示。

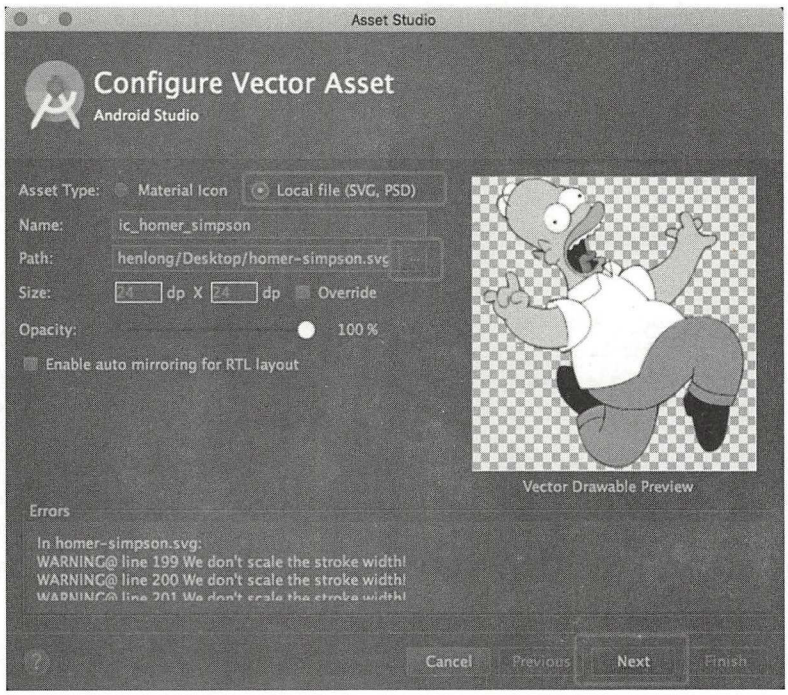


图 7-6　选择 SVG 格式图像的加载路径

接着单击“Next”，即可创建 Vector 图像。在 res 的 drawable 文件夹中，包含创建的 Vector 图像，如 v_homer_simpson.xml。在 Vector 图像中，根属性是 vector 标签，在 vector 标签内排列多行 path 标签。在 vector 的属性中，width 和 height 是矢量图的宽和高，viewportWidth 和 viewportHeight 是画布的宽和高，这些宽和高用于确定图像的绘制坐标。在 path 的属性中，fillColor 是填充的颜色，pathData 是绘制的路径，strokeColor 是线的颜色，strokeLineCap 是线

头尾的形状，strokeWidth 是线的宽度。

```
<vector
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:width="24dp"
    android:height="24dp"
    android:viewportHeight="500.0"
    android:viewportWidth="500.0">
    <path android:fillColor="#00000000"
        android:pathData="M213.1,141.4c0,0 -9.4,3.8 -10.5,7"
        android:strokeColor="#010101"
        android:strokeLineCap="round"
        android:strokeWidth="1.5"/>
    <!--...-->
</vector>
```

将转换完成的 Vector 图像放入布局 activity_image 中展示，与其他的图像相比，几乎没有什么不同，不过分辨率更加优质，效果如图 7-7 所示。

图 7-7　转换完成的 Vector 图像

除了在 Android Studio 中转换 Vector 图像，也可以使用在线网站转换

SVG 转换网址为 http://inloop.github.io/svg2android/。

上传 SVG 格式的图片，就会生成 Vector 图像的代码，注意勾选如图 7-8 所示两个复选框：

◎ Remove empty groups without attributes，即删除没有属性的空组；

◎ Enable support for clip-path (work in progress)，即支持剪切路径；

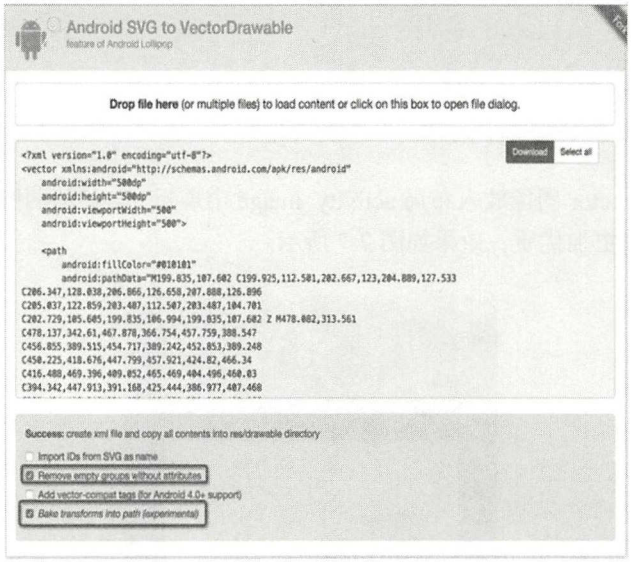


图 7-8　勾选两个复选框

转换完成之后，可以直接下载在线图片，也可以在 res 的 drawable 文件夹中创建 Vector 图像的.xml 文件，粘贴代码。在线生成 Vector 图像的使用方式与 IDE 生成的类似，即与普通图像的使用方式完全相同。

通过观察发现，同为 500×500 分辨率的 PNG 图像大约有 700KB，而 SVG 图像大约有 20KB，仅为原图像的 3%左右，非常节约内存资源。不仅如此，SVG 图像还拥有缩放不变性，可以保持当前的清晰度的情况，继续适配更大的位置。因此，对于一些较大的应用而言，使用 SVG 图像技术处理图片，具有很大的吸引力，可以大大降低应用对于内存的占用。

7.2.2　Vector 动画

通过 SVG 的图像技术，不仅支持创建 Vector 静态图像，还支持创建动态图像，类似于 GIF 格式的效果。注意，动态图像需要最低支持库（Android Support Library）的版本达到 23.2 及以

上。在 Android 的 SDK Manager 中，即可更新版本，如图 7-9 所示。

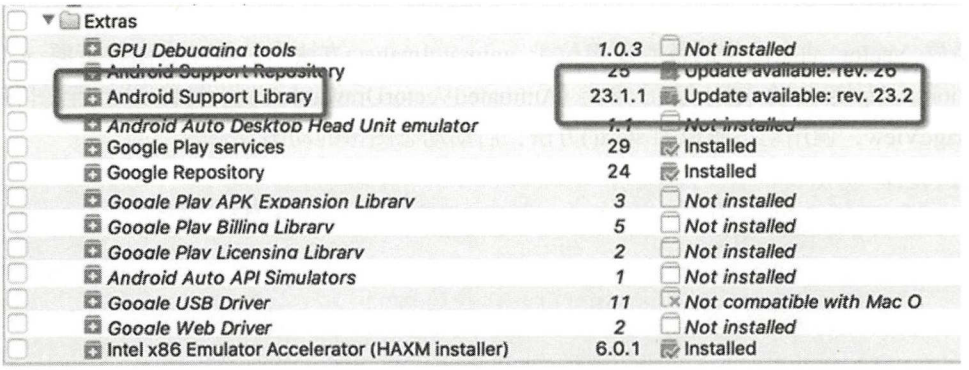


图 7-9　在 SDK Manager 中更新版本

创建 Vector 动态图像主要分为三步：

（1）创建 Vector 静态图像。

（2）创建 Vector 图像的路径属性动画。

（3）将静态图像和属性动画结合到一起，创建 Vector 动态图像（Animated-Vector）；

1. 工程配置

在工程已有的页面 MainActivity 中，添加一个跳转按钮 mBAnimation，跳转至用于显示 Vector 动态图像的页面 AnimationActivity。

```
mBAnimation.setOnClickListener(new View.OnClickListener() {
    @Override public void onClick(View v) {
        startActivity(new Intent(MainActivity.this, AnimationActivity.class));
    }
});
```

在页面 AnimationActivity 的入口函数 onCreate()中，首先设置布局资源 activity_animation，并且使用 ButterKnife 绑定布局 ID；其次播放 Vector 动态图像；最后设置按钮的点击事件，当点击按钮时，重新播放 Vector 动态图像。

```
@Override protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_animation);
    ButterKnife.bind(this);
    animateImage(); // 播放动态图片
    mBDraw.setOnClickListener(new View.OnClickListener() {  // 重绘动画
        @Override public void onClick(View v) {
            animateImage();
```

```
            }
        });
    }
```

播放 Vector 动态图像的核心逻辑位于 animateImage()方法中，将动态图像资源 v_heard_animation，转换为动画向量图像类 AnimatedVectorDrawable，将图像添加至图像控件 mIvImageView，调用动态图像的 start()方法，启动动态图像的动画效果。

```
    private void animateImage() {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
            // 获取动画效果
            AnimatedVectorDrawable mAnimatedVectorDrawable = (AnimatedVectorDrawable)
                    ContextCompat.getDrawable(getApplication(),
R.drawable.v_heard_animation);
            mIvImageView.setImageDrawable(mAnimatedVectorDrawable);
            if (mAnimatedVectorDrawable != null) {
                mAnimatedVectorDrawable.start();
            }
        }
    }
```

下面接着介绍 Vector 动态图像，如 v_heard_animation，是如何被创建的。

2. 动态图像

首先，在 drawable 文件夹中创建 Vector 图像资源 v_homer_simpson.xml，通过 SVG 格式的图片，可以转换为 Vector 图像资源，可以使用 Android Studio 的转换工具，也可以使用在线转换工具。在 Vector 图像的 path 标签中，需要声明 name 属性，用于动画图像的路径选择。Vector 图像资源，如下：

```
    <vector xmlns:android="http://schemas.android.com/apk/res/android"
        android:width="256dp"
        android:height="256dp"
        android:viewportHeight="70"
        android:viewportWidth="70">
        <path
            android:name="heart1"
            android:pathData="..."
            android:strokeColor="#E91E63"
            android:strokeWidth="1"/>
        <path
            android:name="heart2"
            android:pathData="..."
            android:strokeColor="#E91E63"
            android:strokeWidth="1"/>
    </vector>
```

接着，在 animator 文件夹中创建 ObjectAnimator 属性动画 heart_animator.xml，控制绘制状态的变化。属性动画的 trimPathEnd 属性决定绘制路径的长短，其取值区间是从 0~1，即 valueFrom 和 valueTo。如果是从 0~1，则表示全部绘制，如果是从 0.5~1，则表示直接绘制半个，再继续绘制，依此类推。duration 属性表示持续时间为 6000 毫秒（6 秒）。

```
<?xml version="1.0" encoding="utf-8"?>
<objectAnimator
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:duration="6000"
    android:propertyName="trimPathEnd"
    android:valueFrom="0"
    android:valueTo="1"
    android:valueType="floatType"/>
```

最后，在 drawable-v21 中创建 Vector 动态图像 v_heard_animation.xml，表示动态图像只能应用于 Android SDK 21 版本以上。动态图像的根标签是 animated-vector，将 drawable 属性设置为 Vector 图像 v_heard，即动画所需绘制 Vector 图像。在标签 animated-vector 中，添加多个 target 标签，用于控制多个 Vector 图像路径的绘制。target 标签含有 name 属性和 animation 属性，把 Vector 图像资源的路径名，即 path 标签的 name 属性，与路径的属性动画（objectAnimator）组合在一起，用于控制动态图像的绘制。

```
<animated-vector
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:drawable="@drawable/v_heard">
    <target
        android:name="heart1"
        android:animation="@animator/heart_animator"/>
    <target
        android:name="heart2"
        android:animation="@animator/heart_animator"/>
    ...
</animated-vector>
```

把 Vector 动态图像放置入图片控件（ImageView）中，调用 start()方法，即可执行动画效果。动画效果是随着时间绘制图像路径，一直到绘制完成，如图 7-10 所示。

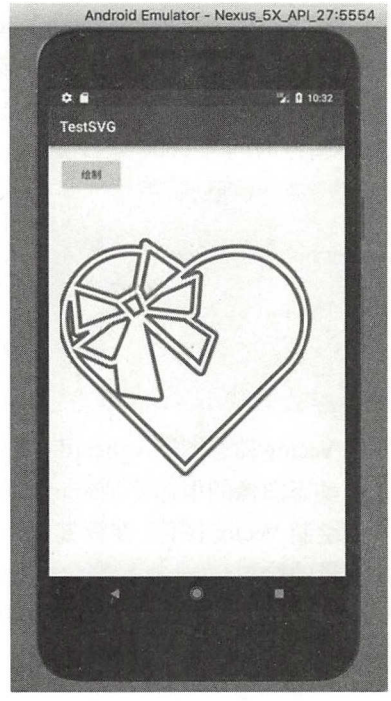


图 7-10　动画效果

7.2.3　第三方 Sharp 库

在 Android 应用中，除了将 SVG 格式图像转换为 Vector 图像才能使用之外，还可以依赖第三方开源库，直接应用 SVG 格式的图片。本节介绍一个有趣的库——Sharp，可用于直接展示 SVG 格式的图像，还支持一些图像的变换。

Sharp 库：https://github.com/Pixplicity/sharp

1. 工程配置

在工程已有的页面 MainActivity 中，再次添加一个跳转按钮 mBSharp，跳转至用于显示 SVG 格式图像的页面 OtherActivity，功能基于第三方开源库 Sharp。

```
mBSharp.setOnClickListener(new View.OnClickListener() {
    @Override public void onClick(View v) {
        startActivity(new Intent(MainActivity.this, OtherActivity.class));
    }
});
```

最终，首页 MainActivity 包含三个跳转按钮，第一个用于跳转显示 Vector 图像的页面，第

二个用于跳转显示 Vector 动画的页面，第三个用于跳转显示 SVG 格式图像的页面。

接着，将 Sharp 的 aar 包放入项目的依赖库中，即 App 目录下的 build.gradle 文件，最新的版本号可以参考 Sharp 的 GitHub 说明。

```
dependencies {
    // ...
    compile 'com.pixplicity.sharp:library:1.1.0@aar' // Sharp 库
}
```

这样，就可以在代码中直接引用开源库 Sharp 了。

2. SVG 图像

在页面 OtherActivity 的入口函数 onCreate()方法中，首先填充布局 activity_other 和使用 ButterKnife 绑定布局 ID；接着调用 Sharp 类的 loadResource()方法加载 SVG 格式图像 cartman.svg，并且将 mSharp 类绑定图片控件 mIvImage，即可在图片控件中显示 SVG 图像；最后设置按钮 mBChange 的点击事件，当点击按钮时，可以修改 SVG 图像的若干属性。

```
@Override protected void onCreate(@Nullable Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_other);
    ButterKnife.bind(this);
    // 加载南方公园的卡通人物
    mSharp = Sharp.loadResource(getResources(), R.raw.cartman);
    mSharp.into(mIvImage);

    // 切换 SVG 的颜色
    mBChange.setOnClickListener(new View.OnClickListener() {
        @Override public void onClick(View v) {
            changeSVG();  // 修改 SVG 的图片
        }
    });
}
```

其中，`cartman.svg` 是标准的 SVG 格式图像，类似于 XML 格式，其中重要的标签是路径 path，用于绘制图像的各个模块。通过设置 path 标签的 ID 属性，可以在 Sharp 中修改 SVG 格式图像的 path 标签，变换图像的效果。

```
<svg>
  <metadata>
  <!--...-->
  </metadata>
  <defs/>
  <sodipodi:namedview/>
  <path
```

```
        d="M14,85l3,9h72c0,0,5-9,4-10c-2-2-79,0-79,1"
        fill="#7C4E32"
        id="pants" />
    <!--...-->
</svg>
```

在 changeSVG()方法中含有两段逻辑，一段是随机修改 SVG 图像的配色，一段是在按钮中创建尺寸较小的 SVG 图像。

```
private void changeSVG() {
    mSharp.setOnElementListener(new OnSvgElementListener() {}); // 修改配色
    mSharp.into(mIvImage); // 填充控件
    mSharp.getSharpPicture(new Sharp.PictureCallback() {}); // 创建较小尺寸图像
}
```

首先，调用 Sharp 的 setOnElementListener()接口，可以改变 SVG 图像的元素属性，接口监听 SVG 图像的启动绘制、终止绘制、元素绘制、元素绘制完成四个状态，即 onSvgStart()、onSvgEnd()、onSvgElement()、onSvgElementDrawn()。改变 SVG 图像的元素属性需要实现 onSvgElement()方法，根据 SVG 图像 Path 路径的 ID 属性，如果是 T 恤（shirt）、帽子（hat）和裤子（pants），则重新并且随机（Random）设置画笔（paint）的颜色，则这三个部分的颜色在每次绘制时都是随机的，会不断地变换，最后返回元素泛型 element。

```
mSharp.setOnElementListener(new OnSvgElementListener() {
    @Override public void onSvgStart(@NonNull Canvas canvas, @Nullable RectF
rectF) {}
    @Override public void onSvgEnd(@NonNull Canvas canvas, @Nullable RectF rectF)
{}
    @Override public <T> T onSvgElement(
            @Nullable String id, @NonNull T element, @Nullable RectF
elementBounds,
            @NonNull Canvas canvas, @Nullable RectF canvasBounds, @Nullable Paint
paint) {
        if (("shirt".equals(id) || "hat".equals(id) || "pants".equals(id))) {
            Random random = new Random();
            if (paint != null) {
                paint.setColor(Color.argb(255,
                        random.nextInt(256), random.nextInt(256), random.nextInt
(256)));
            }
        }
        return element;
    }
    @Override public <T> void onSvgElementDrawn(
        @Nullable String s, @NonNull T t, @NonNull Canvas canvas, @Nullable Paint
paint) {}
});
```

接着，将 Shape 类注入至图片控件 ImageView 中，即可显示，每次都会显示不同配色的 SVG 图像。

```
mSharp.into(mIvImage);
```

除此之外，调用 Shape 的 getSharpPicture()接口，覆写图片准备方法 onPictureReady()。首先，获取需要设置图片的像素 size；其次，调用 SharpPicture 的 createDrawable()方法，创建可适配按钮 mBChange 大小的图像 drawable；最后，将图像 drawable 放置于按钮 mBChange 的左侧。由于当创建 SVG 图像时，已经覆写 mSharp 的元素绘制的方法，因此每次的图像配色也是随机产生的。

```
mSharp.getSharpPicture(new Sharp.PictureCallback() {
    @Override public void onPictureReady(SharpPicture sharpPicture) {
        int size = getResources().getDimensionPixelSize(R.dimen.icon_size);
        Drawable drawable = sharpPicture.createDrawable(mBChange, size);
        mBChange.setCompoundDrawables(drawable, null, null, null);
// 设置左侧图像
    }
});
```

最终的效果是当每次点击按钮时，都会随机生成不同配色的 SVG 图像，如图 7-11 所示。

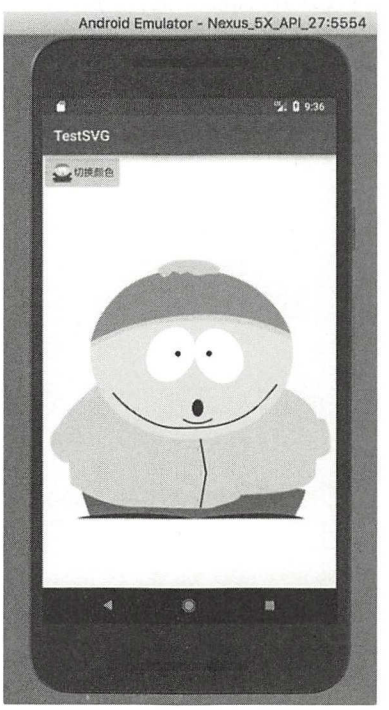


图 7-11　随机生成不同配色的 SVG 图像

对于 SVG 格式的图像的优势，本文已经列举了很多，其特性就是对于简单的矢量图像使用编码方式，而不是像素方式存储。其优势性就在于编码方式，其优点有：

◎ 编码方式支持批量绘制像素点，相比存储单个点，具有磁盘占用较小的优势。

◎ 编码方式支持等比例扩展，较小的图像适配较大的空间，也不会出现过多的损耗。

◎ 编码方式支持时间控制绘制，当绘制线条动画时，不用存储多幅图片，节省性能和内存。

◎ 编码方式支持动态控制绘制，当绘制区域时直接修改颜色等属性，不必重新构图。

由此可见，对于在移动应用中常见的、简单的矢量图，都可以使用 SVG 图像进行处理，具体转换方式交给视觉设计的同事们完成即可，开发人员要做的是将转换完成的 SVG 图像集成于应用之中。但是，SVG 格式并不是万能的，主要的缺点是，不适应于细节过多的图像，如风景画、照片等，这些图像需要转换的像素块过于密集，占用存储空间反而较大，SVG 格式的优势在于处理简单的色块。

在 Android 中，SVG 格式图像主要的表现形式就是 Vector 图像。本文已经系统地分析静态和动态 Vector 图像的使用方式和注意细节，还介绍了一个支持直接应用 SVG 图像的开源库 Sharp，这些已经可以指导，如何在 Android 开发中使用 SVG 格式的图像。对于高级程序员而言，时刻注意技术的革新与变化，才能为自己的产品注入更多的活力，开发出更加优秀的产品。

第8章 测试与优化

8.1 基于 Espresso 和 Dagger 的自动化测试框架

在任何线上项目中，测试都是至关重要、不可取代的一个步骤，Android 项目同样如此，甚至会更加重要。因为每一次迭代开发的 Android 项目都会添加新的功能，需要发布新的版本。一般而言，应用的主要下载渠道是各个应用市场，国内常见的有应用宝、360 应用市场、小米应用商店等。如果直接把最终生成的应用包交给第三方平台，一旦出了问题就会非常麻烦，只能再次发布版本。发版之后，还需要通知用户重复更新，非常影响用户体验，让用户对于应用的品质产生怀疑。虽然已经出现很多较为成熟的 HotFix 技术框架，但是只能修复一些细微的错误，对于较为严重或者深层次的错误仍然无能为力，还是需要发布版本解决。因此，Android 的功能测试是做好应用的必备保障。

对于可靠的功能测试而言，意味着在任何状态下获取的测试结果都需要一致，因此模拟数据的环节必不可少。对于测试工具而言，官方推荐使用 Espresso；对于模拟数据而言，可以使用依赖注入工具 Dagger。目前，Espresso 和 Dagger 都由 Google 团队进行维护，保证库的质量和一致性，基于 Espresso 和 Dagger 也是比较主流的自动化测试框架。

```
    Espresso : https://google.github.io/android-testing-support-library/docs/
espresso/
    Dagger: https://google.github.io/dagger/
```

除了测试之外，Dagger 已经是主流的依赖注入框架，最初由 Square 公司组织开发，并且针对 Android 进行特别地优化，现在已经被 Google 公司团队接手，进行统一迭代。Dagger 也是众

多 Android 开发者的必备工具之一，不像其他的依赖注入工具，Dagger 并没有使用反射，而是使用预生成代码的形式提高运行时的执行速度。

在传统的测试中，一般会模拟所有的依赖，保证在相同状态下输出数据的一致性，避免出现测试不可靠的情况。而基于 Espresso 和 Dagger 的自动化测试框架，使得 Android 的功能测试也可以实现类似的效果，即 Espresso 负责测试，Dagger 负责数据模拟。本例通过创建一个经典的显示日期应用，模拟返回稳定的网络数据，实现远程访问功能的覆盖测试。当然，这个数据模拟框架也支持测试其他不稳定的数据访问，如数据库、本地文件等。

本文实例完整代码的下载地址为 https://github.com/SpikeKing/wcl-espresso-dagger-demo。

8.1.1 工程配置

为了测试显示日期的功能，首先创建一个简易的 Hello World 工程，并且修改 App 目录下的工程配置 build.gradle，添加功能和测试需要依赖的第三方库，主要包含以下几类：

◎ 支持 Lambda 表达式功能。
◎ 支持静态编译注解功能。
◎ 支持 DataBinding 数据绑定功能。
◎ 支持 Dagger 依赖注入功能。
◎ 支持 RxJava 响应式编程功能。
◎ 支持 Retrofit 网络访问功能。
◎ 支持 Espresso 测试功能。

Lambda 表达式用于支持省略匿名类，简化代码编写，本例使用 Retrolambda 插件，并且设置编译选项支持 Java 1.8 版本。

```
plugins {
    id "me.tatarka.retrolambda" version "3.2.4"
}

android {
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
}
```

静态编译注解，即 Android-Apt 包，为将依赖注入所需的注解编译为静态文件提供支持，在编译时处理注解逻辑，避免在运行时处理，提升执行性能。

```
buildscript {
```

```
        // ...
        dependencies {
            classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'
        }
    }

    apply plugin: 'com.neenbedankt.android-apt' // 处理注解
```

DataBinding 数据绑定，用于绑定布局中的资源，直接通过修改数据类的属性，即可修改页面的展示效果。

```
    dataBinding {
        enabled = true
    }
```

Dagger 依赖注入用于注入网络数据，在测试时也可用于注入模拟的网络数据。将类的实例化与业务逻辑解耦，即在不修改业务逻辑的情况下，实例化为任意子类，用于测试或其他需求。

```
    final DAGGER_VERSION = '2.0.2'

    dependencies {
        compile "com.google.dagger:dagger:${DAGGER_VERSION}" // dagger2
        compile "com.google.dagger:dagger-compiler:${DAGGER_VERSION}" // dagger2
    }
```

RxJava 响应式编程用于处理异步逻辑，避免大量回调的冗余，和简化常用功能的开发。RxAndroid 是 RxJava 的扩展，在其之上，额外添加若干个 Android 线程的支持。

```
    dependencies {
        compile 'io.reactivex:rxandroid:1.1.0' // RxAndroid
    }
```

Retrofit 网络访问用于处理 REST 网络访问，简化网络请求的 URL 设置、异步处理、数据解析等工作。异步处理的适配器使用 adapter-rxjava，即 RxJava；数据解析的转换器使用 converter-gson，即 Gson。

```
    final RETROFIT_VERSION = '2.0.0-beta2'

    dependencies {
        compile "com.squareup.retrofit:retrofit:${RETROFIT_VERSION}"
    // Retrofit 网络处理
        compile "com.squareup.retrofit:adapter-rxjava:${RETROFIT_VERSION}"
    // Retrofit 的 rx 解析库
        compile "com.squareup.retrofit:converter-gson:${RETROFIT_VERSION}"
    // Retrofit 的 gson 库
        compile 'com.squareup.okhttp:logging-interceptor:2.6.0' // 拦截器
    }
```

Espresso 测试用于功能测试，支持测试页面逻辑，提供测试开发范式简化开发流程，依赖

Android JUnit Runner 测试库和 JUnit4 Rules 测试库。

```
    dependencies {
        androidTestCompile 'com.android.support.test:runner:0.4.1' // Android JUnit
Runner
        androidTestCompile 'com.android.support.test:rules:0.4.1' // JUnit4 Rules
        androidTestCompile
'com.android.support.test.espresso:espresso-core:2.2.1' // Espresso core
    }
```

其余的 build.gradle 配置信息保持不变。

Retrolambda 插件、Android-Apt 包、DataBinding 功能、Dagger 库、RxJava 库、Retrofit 库和 Espresso 库都是非常主流的、常见的开发库，可以大大地提高我们的开发效率，在大型工程中都会普遍使用。

8.1.2 业务逻辑

本例实现了一个非常简单的天气预报功能，输入城市的名称，在第三方天气预报数据源中获取城市的天气预报。

1. 布局

首先，使用 DataBinding 模式创建页面布局 activity_main.xml。DataBinding 模式是以 layout 标签为根布局，其中含有数据部分，即 data 标签，以及布局部分，如 FrameLayout 标签。在数据部分 data 标签中，含有变量 variable 标签，用于存储与布局绑定的对象，name 属性表示名称，type 属性表示类对象。在布局部分中，控件可以直接使用类的变量值，如 @{weatherData.temperatureCelsius}，设置控件的属性。本例的布局框架，大致如下：

```
    <layout>
        <data>
            <variable
                name="weatherData"

type="org.wangchenlong.wcl_espresso_dagger_demo.data.WeatherData"/>
        </data>
        <FrameLayout>
            <RelativeLayout>
                <TextView
                    android:id="@+id/temperature"
                    android:text="@{weatherData.temperatureCelsius}"
                    tools:text="10°"/>
                <TextView/>
                <TextView/>
```

```
            <TextView/>
            <TextView/>
            <LinearLayout>
                <TextView/>
                <TextView/>
            </LinearLayout>
        </RelativeLayout>
        <ProgressBar
            android:id="@+id/progress"/>
    </FrameLayout>
</layout>
```

关于 DataBinding 的数据部分 WeatherData，就是简单的 POJO(Plain Old Java Object)对象，提供数据的设置和访问接口。首先，声明常量，即日期格式 DATE_FORMAT、Kalvin 零度 KELVIN_ZERO、摄氏度格式 FORMAT_TEMPERATURE_CELSIUS 等；其次，声明变量，即城市名称 name、天气数组 weather、气象信息 main 等；再次，提供数据的访问接口，即获取城市名称 getCityName()，获取日期 getWeatherDate()、获取气象信息 getWeatherState()、获取天气描述 getWeatherDescription()、获取摄氏度 getTemperatureCelsius()、获取湿度 getHumidity()等；最后，声明两个内部类，即天气信息类 Weather、气象信息类 Main。注入网络数据 Json 的方式选用 Gson 形式，因此变量的名称需要与网络数据 Json 的关键词（Key）一致，不需要提供 set() 方法或 public 访问权限，也可以设置数据。

```
public class WeatherData {  // 日期对象
    public static final String DATE_FORMAT = "EEEE, d MMM";  // 日期格式
    private static final int KELVIN_ZERO = 273;  // 开尔文 0 度
    private static final String FORMAT_TEMPERATURE_CELSIUS = "%d°";

    private String name; // 城市名称
    private Weather[] weather; // 天气数组
    private Main main; // 气象信息

    public String getCityName() {  // 获取名称
        return name;
    }
    public String getWeatherDate() {  // 获取日期
        return new SimpleDateFormat(DATE_FORMAT, Locale.getDefault()).format
(new Date());
    }
    public String getWeatherState() {  // 获取气象信息
        return weather().main;
    }
    public String getWeatherDescription() {  // 获取天气描述
        return weather().description;
    }
    public String getTemperatureCelsius() {  // 获取摄氏度
```

```
            return String.format(FORMAT_TEMPERATURE_CELSIUS, (int) main.temp -
KELVIN_ZERO);
        }
        public String getHumidity() { // 获取湿度
            return String.format("%d%%", main.humidity);
        }
        private Weather weather() { // 返回天气
            return weather[0];
        }
        private static class Weather { // 天气信息类
            private String main; // 简介
            private String description; // 描述
        }
        private static class Main { // 气象信息类
            private float temp; // 温度
            private int humidity; // 湿度
        }
    }
```

重新构建工程，自动创建 ActivityMainBinding 类，用于在页面中设置和访问布局中的变量（如 WeatherData）和控件。

2．网络

天气预报的信息源来自于 OpenWeatherMap，HRL 网址为 https:// openweathermap .org/，感谢 OpenWeatherMap 提供的开放数据。

首先，创建日期 API 客户端接口 WeatherApiClient，访问的主页 END_POINT 是“http://api.openweathermap.org/data/2.5/”，Get 域是“weather”，Query 参数是“q”，使用 Observable 格式处理返回的异步数据。

```
    public interface WeatherApiClient {
        String END_POINT = "http://api.openweathermap.org/data/2.5/";
        @GET("weather") Observable<WeatherData> getWeatherForCity(@Query("q")
String cityName);
    }
```

如果 q 的值是“Beijing”，访问的网址类似于

http://api.openweathermap.org/data/2.5/weather?q=beijing。

接着，创建 API 关键字拦截器 ApiKeyInterceptor，核心接口是拦截方法 intercept()，获取链接 Chain 中的请求 Request，获取请求类的 HttpUrl 链接，调用 addQueryParameter()在链接中添加参数 QUERY_APP_ID，创建新的 HttpUrl，再创建新的请求 Request，将新的请求放入已有的链接 Chain 中。

```
    public class ApiKeyInterceptor implements Interceptor {
```

```
        private static final String QUERY_APP_ID = "APPID";  // 用户 ID
        private final String mApiKey;  // 用户 Key
        public ApiKeyInterceptor(String apiKey) {
            mApiKey = apiKey;
        }
        @Override public Response intercept(Chain chain) throws IOException {
            Request request = chain.request();
            HttpUrl url = request.httpUrl().newBuilder().addQueryParameter(QUERY_
APP_ID, mApiKey).build();
            Request newRequest = chain.request().newBuilder().url(url).build();
            return chain.proceed(newRequest);
        }
    }
```

其功能是在 URL 链接中添加常量参数。一般而言，可变参数放入网络请求客户端，如 WeatherApiClient，常量参数放入拦截器，如 ApiKeyInterceptor。如果 APPID 的值是“abcd”，访问的网址类似于

http://api.openweathermap.org/data/2.5/weather?q=beijing&APPID=abcd。

APPID 在 OpenWeatherMap 的网站上申请，网站用于标志来访用户，是安全管理的一部分，目前免费申请。只有含有 APPID 的请求，才能获取访问的日期数据。

3．注入

本例采用依赖注入的方式，将网络访问模块注入至业务逻辑中。

首先，依赖注入最基本的部分是模块，模块 AppModule 提供两个注入类，一个是应用的上下文 Context，一个是网络访问客户端 WeatherApiClient。其中，上下文 Context 被标注为 @AppScope，表明属于运行时注入。网络访问客户端 WeatherApiClient 是模块的核心。

在 WeatherApiClient 中，首先，创建 OkHttp 的客户端 OkHttpClient；其次，在客户端中加入两个拦截器，一个是访问地址的 API Key，另一个是 HTTP 日志功能；再次，创建 Retrofit 网络请求类，在其中添加 URL 地址，添加 RxJava 的异步返回数据适配器、添加 Gson 的 Json 数据转换器、添加 OkHttp 客户端；最后，Retrofit 网络请求类实现 WeatherApiClient 的接口，完成网络服务逻辑的创建。

```
    @Module public class AppModule {
        private final Context mContext;
        public AppModule(Context context) {
            mContext = context;
        }
        @Provides @AppScope public Context provideAppContext() {
            return mContext;
        }
        @Provides public WeatherApiClient provideWeatherApiClient() {
```

```
        OkHttpClient client = new OkHttpClient();
        client.interceptors().add(
                new  ApiKeyInterceptor(mContext.getString(R.string.open_weather
_api_key)));
        client.interceptors().add(new HttpLoggingInterceptor());
        return new Retrofit.Builder()
                .baseUrl(WeatherApiClient.END_POINT)
                .addCallAdapterFactory(RxJavaCallAdapterFactory.create())
                .addConverterFactory(GsonConverterFactory.create())
                .client(client)
                .build()
                .create(WeatherApiClient.class);
    }
}
```

关于运行时注入的注解@AppScope。由于 Context 是在运行时才能被获取的信息，因此在声明注入方法时需要添加运行时注解。@AppScope 的实现逻辑如下：

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
public @interface AppScope {
}
```

接着，将模块 AppModule 放入组件接口 AppComponent 中。在 inject()方法中，设置需要注入的页面类 MainActivity。组件也被声明为@AppScope，即运行时注入，因为在模块中，含有运行时注入类 Context。重新构建工程，自动创建 DaggerAppComponent 类，即在静态编译时提供注入接口。

```
@AppScope
@Component(modules = AppModule.class)
public interface AppComponent {
    void inject(MainActivity activity);
}
```

最后，将组件 AppComponent 放置于全部页面都支持访问的类 Application 中。WeatherApplication 继承于 Application，覆写入口函数 onCreate()，创建 AppComponent 组件，添加 AppModule 模块，同时，提供 AppComponent 组件的访问接口。

```
public class WeatherApplication extends Application {
    private AppComponent mAppComponent;

    @Override public void onCreate() {
        super.onCreate();
        mAppComponent = DaggerAppComponent.builder()
                .appModule(new AppModule(this))
                .build();
    }
```

```
    public AppComponent getAppComponent() {
        return mAppComponent;
    }
}
```

4．逻辑

布局与网络客户端已经准备完成，接下来是在页面 MainActivity 中编写核心的业务逻辑，实现简单的搜索城市的天气信息功能。

在页面 MainActivity 中，首先声明变量，即 DataBinding 类 mBinding（自动创建）、菜单项 mSearchItem、网络数据订阅源 mSubscription、天气客户端 mWeatherApiClient（注入）。在入口函数 onCreate()中，第一步，获取 Application 类，再获取 AppComponent 组件，将组件注入至当前页面（this）中；第二步，调用 DataBindingUtil 的 setContentView()方法，绑定 MainActivity 页面和 activity_main 布局，创建 DataBinding 类 mBinding。注意的是，mWeatherApiClient 不需要实例化，由@Inject 直接在页面中注入实例，这个被注入的实例，可以根据注入源的不同随时替换，为网络的功能测试部分提供了方便的接口。

```
    private ActivityMainBinding mBinding; // 数据绑定
    private MenuItem mSearchItem; // 菜单项
    private Subscription mSubscription; // 订阅
    @Inject WeatherApiClient mWeatherApiClient; // 天气客户端

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ((WeatherApplication) getApplication()).getAppComponent().inject(this);
        mBinding = DataBindingUtil.setContentView(MainActivity.this, R.layout.activity_
main);
    }
```

其次，设置搜索逻辑。在搜索框中输入城市名称，即可搜索当前的天气信息。在菜单列表中，填充搜索布局 menu_activity_main，获取搜索布局中的搜索项 mSearchItem。在 tintSearchMenuItem()方法中，修改搜索项的颜色样式。在 initSearchView()中，设置搜索框的监听事件，含有搜索的核心业务逻辑，调用 setOnQueryTextListener()设置搜索框的回调事件：当提交搜索内容时，调用 onQueryTextSubmit()方法，当修改搜索内容时，调用 onQueryTextChange()。重点关注提交搜索信息的操作，调用 MenuItemCompat 的 collapseActionView()方法恢复搜索项的起始状态，同时，调用 loadWeatherData()方法加载远程的天气数据。

```
    @Override public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.menu_activity_main, menu); // 加载目录资源
        mSearchItem = menu.findItem(R.id.menu_action_search);
        tintSearchMenuItem();
        initSearchView();
```

```
        return true;
    }

    private void tintSearchMenuItem() {  // 搜索项着色，会覆盖基础颜色，取交集
        int color = ContextCompat.getColor(this, android.R.color.white); // 白色
        mSearchItem.getIcon().setColorFilter(color, PorterDuff.Mode.SRC_IN);
    // 交集
    }

    private void initSearchView() {  // 搜索项初始化
        SearchView searchView = (SearchView) MenuItemCompat.getActionView(mSearchItem);
        searchView.setOnQueryTextListener(new SearchView.OnQueryTextListener() {
            @Override public boolean onQueryTextSubmit(String query) {
                MenuItemCompat.collapseActionView(mSearchItem);
                loadWeatherData(query); // 加载查询数据
                return true;
            }

            @Override public boolean onQueryTextChange(String newText) {
                return false;
            }
        });
    }
```

搜索框的效果如图 8-1 所示。

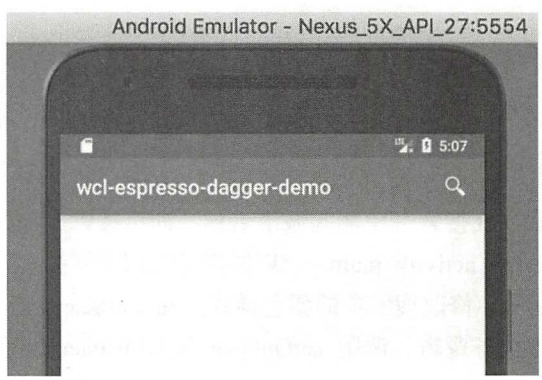


图 8-1　搜索框的效果

最后，加载天气信息的网络数据，即 loadWeatherData()方法，在其中使用 DataBinding 进行控件修改和数据绑定。首先，显示进度条 mBinding.progress，由于网络访问是耗时操作，使用进度条可以提示用户当前操作正在执行中；接着，使用 Retrofit 架构的网络客户端 mWeatherApiClient，在 getWeatherForCity()中根据天气参数 cityName 拼接 URL，在 subscribeOn()中执行网络操作，在 observeOn()中监听网络操作（Android 主线程），当请求成功时，执行

bindData()方法处理数据，当请求失败时，执行 bindDataError()方法处理异常。this::bindData 是 Lambda 表达式的简写形式，即方法引用（Method References），当方法的参数和返回值与匿名类函数相同时，直接由 this::方法名代替。

```
private void loadWeatherData(String cityName) {
    mBinding.progress.setVisibility(View.VISIBLE);
    mSubscription = mWeatherApiClient
            .getWeatherForCity(cityName)
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(this::bindData, this::bindDataError);
}
```

在 bindData()方法中，隐藏进度条 mBinding.progress，显示天气布局 mBinding.weatherLayout，将数据 weatherData 置入 mBinding 中，则数据会自动地在布局的控件中展示。在 bindDataError()中，仅仅隐藏进度条 mBinding.progress，表示网络请求失败。

```
private void bindData(WeatherData weatherData) {
    mBinding.progress.setVisibility(View.INVISIBLE);
    mBinding.weatherLayout.setVisibility(View.VISIBLE);
    mBinding.setWeatherData(weatherData);
}

private void bindDataError(Throwable throwable) {
    mBinding.progress.setVisibility(View.INVISIBLE);
}
```

在 onDestroy()中，当网络请求未执行完成时，为了避免内存泄漏，直接取消网络客户端 mSubscription。

```
@Override
protected void onDestroy() {
    if (mSubscription != null) {
        mSubscription.unsubscribe();
    }
    super.onDestroy();
}
```

如在搜索框中输入“Beijing”，就会展示北京的天气情况，效果如图 8-2 所示。

图 8-2　展示效果

至此，业务逻辑已经开发完成，如何基于 Espresso 和 Dagger 框架进行功能测试，保证网络数据的访问可用性呢？

8.1.3　功能测试

由于自身的问题，网络数据一直是自动化测试的难点之一，主要体现在：

◎　如何针对网络数据源连接的脆弱性，保证测试结果的稳定性？

◎　如何针对网络数据源修改的易变性，保证测试方法的扩展性？

这一切的根源，还是在于测试与业务的耦合过于密切，难以分割，而依赖注入的思想恰好将类的实例与类的声明解耦，让业务层依赖于原有脆弱的、易变的网络数据，而让测试层依赖于模拟稳定的、扩展的网络数据实例。Dagger 正是依赖注入最实用的工具之一，用于解决业务与测试耦合的问题。

功能测试位于“app”/“src”/“androidTest”目录下，与业务源码的包名相同，如图 8-3 所示。

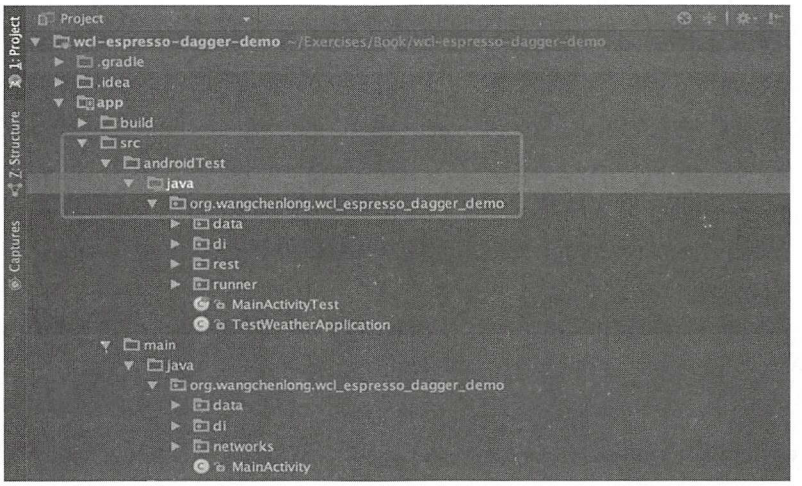


图 8-3　功能测试的位置

1. 测试源

测试源是采用 Dagger 依赖注入的形式，与普通的依赖注入类似，位于 DI（Dependency Injection）文件夹下，结构也是模块和组件形式。首先，创建测试模块 TestAppModule，提供 AppContext 注入和 WeatherApiClient 注入，其中 WeatherApiClient 使用模拟的网络请求客户端类 MockWeatherApiClient，而不是正式的网络请求客户端类 WeatherApiClient。

```
@Module public class TestAppModule {
    private final Context mContext;

    public TestAppModule(Context context) {
        mContext = context.getApplicationContext();
    }

    @AppScope @Provides public Context provideAppContext() {
        return mContext;
    }

    @Provides public WeatherApiClient provideWeatherApiClient() {
        return new MockWeatherApiClient();
    }
}
```

接着，创建测试组件 TestAppComponent，声明注入方法 inject()，支持在 MainActivityTest 类中注入模块 TestAppModule。组件属于运行时注入，即标记@AppScope 注解。重新构建工程，自动创建 Dagger 模板实例类 DaggerTestAppComponent，用于在编译时注入。

```
@AppScope @Component(modules = TestAppModule.class)
```

```
public interface TestAppComponent extends AppComponent {
    void inject(MainActivityTest test);
}
```

最后，设置依赖注入的入口类 TestWeatherApplication，继承于 WeatherApplication。声明测试组件变量 mTestAppComponent。在入口函数 onCreate() 中，根据自动生成的 DaggerTestAppComponent 类创建测试组件，在其中添加 TestAppModule 模块实例。入口类提供测试组件的访问接口 getAppComponent()。至此，在测试环境中，依赖注入的三个部分（即 Module、Component、Application）已经完成。

```
public class TestWeatherApplication extends WeatherApplication {
    private TestAppComponent mTestAppComponent;

    @Override public void onCreate() {
        super.onCreate();
        mTestAppComponent = DaggerTestAppComponent.builder()
                .testAppModule(new TestAppModule(this))
                .build();
    }

    @Override public TestAppComponent getAppComponent() {
        return mTestAppComponent;
    }
}
```

需要注意的是，在测试模块 TestAppModule 中，提供的网络请求 WeatherApiClient 依赖注入，使用模拟 API 类，即返回 MockWeatherApiClient 实例。模拟网络请求类 MockWeatherApiClient 实现原有网络请求接口类 WeatherApiClient，实现 getWeatherForCity()方法。在根据城市获取天气的接口 getWeatherForCity()中，使用 Gson 和测试数据 MUNICH_WEATHER_DATA_JSON 创建模拟的网络数据 WeatherData，并且创建 RxJava 的数据观察源 Observable，延迟 1 秒返回。延迟 1 秒，用于模拟网络中的延迟情况。这样，在测试过程中，不依赖于网络的状态，提供始终如一的网络数据 MUNICH_WEATHER_DATA_JSON。

```
public class MockWeatherApiClient implements WeatherApiClient {
    @Override public Observable<WeatherData> getWeatherForCity(String cityName) {
        WeatherData weatherData = new Gson()
                .fromJson(TestData.MUNICH_WEATHER_DATA_JSON, WeatherData.class);
// 获得模拟数据
        return Observable.just(weatherData).delay(1, TimeUnit.SECONDS);
// 延迟时间
    }
}
```

关于测试的网络数据 TestData，来源于真实请求的返回数据，格式为 Json，保存在字符串中，用于测试网络功能。

```
public final class TestData {
    public static final String MUNICH_WEATHER_DATA_JSON = "\n" +
            "{\n" +
            "    \"coord\": {\n" +
            "        \"lon\": 11.58,\n" +
            "        \"lat\": 48.14\n" +
            "    },\n" +
            // ...
            "}";
    private TestData() {
        // no instances
    }
}
```

2. 测试逻辑

在基于 Espresso 和 Dagger 的自动化测试框架中，第一步是基于 Dagger 的测试源已经准备完成，第二步就是基于 Espresso 的测试逻辑。

首先，测试运行器 WeatherTestRunner 继承于 AndroidJUnitRunner，覆写 newApplication() 接口。 在 newApplication()中，根据反射的原理在测试环境中添加测试 Application，获取 TestWeatherApplication 的类名，使用类加载器 ClassLoader 根据类名创建类实例。JUnit 是 Java 单元测试框架，由于 Android 是基于 Java 语言进行开发的，同样地，测试框架也是基于 JUnit 框架。

```
public class WeatherTestRunner extends AndroidJUnitRunner {
    @Override
    public Application newApplication(ClassLoader cl, String className, Context
context) throws InstantiationException,
            IllegalAccessException, ClassNotFoundException {
        String                    testApplicationClassName                    =
TestWeatherApplication.class.getCanonicalName();
        return super.newApplication(cl, testApplicationClassName, context);
    }
}
```

在工程配置 build.gradle 中，声明测试运行器 WeatherTestRunner，以保证在测试环境中可以执行。在默认配置 defaultConfig 中，设置 testInstrumentationRunner 属性，将 WeatherTestRunner 的完整路径以字符串的形式放入其中。

```
android {
    defaultConfig {
        testInstrumentationRunner "org.wangchenlong.wcl_espresso_dagger_demo.
runner.WeatherTestRunner"
    }
}
```

接着，在测试主类 MainActivityTest 中，编写功能测试的核心逻辑。测试类一般与被测试类的名称相似，即在被测试类的名称后面添加“Test”，便于辨识，如 MainActivityTest 是 MainActivity 的测试类。

◎ 注解@LargeTest，表示大型测试，支持的测试功能较为丰富，同样地，还有@SmallTest、@MediumTest，支持的测试功能较少。

◎ 注解@RunWith(AndroidJUnit4.class)，表示运行于 AndroidJUnit4 框架，同时加载测试的 Application 类。

◎ 变量 CITY_NAME，用于测试的输入，不过由于测试数据的一致性，输入任何城市数据均不变。

◎ 变量 activityTestRule 将 MainActivity.class 封装为测试类 ActivityTestRule，用于在泛型类中包装测试相关的功能。

◎ 变量 weatherApiClient，通过@Inject 注入 WeatherApiClient 的实例，即含有测试数据的网络服务类。

```
@LargeTest @RunWith(AndroidJUnit4.class)
public class MainActivityTest {
    private static final String CITY_NAME = "Beijing"; // 任何城市数据均不变
    @Rule public ActivityTestRule<MainActivity> activityTestRule =
            new ActivityTestRule<>(MainActivity.class);
    @Inject WeatherApiClient weatherApiClient;  // 测试服务

}
```

在测试主类 MainActivityTest 中，在功能测试执行前，调用 setUp()方法，进行测试的初始化设置。注解@Before 表示此方法在功能测试前执行。在 setUp()方法中，调用 activityTestRule 的 getActivity()获取 MainActivity 页面类，调用 getApplication()获取 TestWeatherApplication 测试应用类，调用 getAppComponent()获取组件，最后调用 inject()，将依赖注入至 MainActivityTest 类（this）中。

```
@Before public void setUp() {
    Log.e("WCL-TEST", "setUp");
    ((TestWeatherApplication)
activityTestRule.getActivity().getApplication())
            .getAppComponent().inject(this);
}
```

功能测试的逻辑位于 correctWeatherDataDisplayed()中，通过方法名的含义可知，此方法的功能是验证天气数据是否正常显示。第一步，模拟用户的操作，即 perform()方法：

（1）单击（click）搜索按钮（menu_action_search）。

（2）在搜索文本框（search_src_text）中输入（replaceText）城市名称 CITY_NAME。

（3）在搜索文本框（search_src_text）中按压（pressKey）键盘的确认键 KEYCODE_ENTER。

完成搜索城市天气的操作，输出和展示的数据就是已注入的测试网络数据源 WeatherApiClient，即 MockWeatherApiClient 实例。

第二步，验证输出的数据，即 check()方法：

（1）获取测试网络数据 weatherData，调用 getWeatherForCity(CITY_NAME)获取 RxJava 异步观察者，调用 toBlocking()获取异步返回数据，调用 first()获取正确时网络数据，即第一个部分。RxJava 的返回数据含有三个部分，第一部分是正确时网络数据，第二部分是错误时网络数据，第三部分是异常时的网络数据。

（2）显示的城市名称 city_name，与测试数据的城市名称（getCityName()）一致。

（3）显示的天气日期 weather_date，与测试数据的天气日期（getWeatherDate()）一致。

（4）显示的天气状态 weather_state，与测试数据的天气状态（getWeatherState()）一致。

（5）显示的天气描述 weather_description，与测试数据的城市描述（getWeatherDescription()）一致。

（6）显示的温度 temperature，与测试数据的摄氏温度（getTemperatureCelsius()）一致。

（7）显示的湿度 humidity，与测试数据的湿度（getHumidity()）一致。

完成验证显示数据的操作，保证显示数据与数据源数据一致。

```
    @Test public void correctWeatherDataDisplayed() {
        Log.e("WCL-TEST", "correctWeatherDataDisplayed");
        // 执行操作
        onView(ViewMatchers.withId(R.id.menu_action_search)).perform(click());

onView(withId(android.support.v7.appcompat.R.id.search_src_text)).perform(replac
eText(CITY_NAME));

onView(withId(android.support.v7.appcompat.R.id.search_src_text)).perform(pressK
ey(KeyEvent.KEYCODE_ENTER));

        // 验证逻辑
        WeatherData weatherData = weatherApiClient.getWeatherForCity(CITY_NAME).
toBlocking().first();

onView(withId(R.id.city_name)).check(matches(withText(weatherData.getCityName())
));

onView(withId(R.id.weather_date)).check(matches(withText(weatherData.getWeatherD
ate())));

onView(withId(R.id.weather_state)).check(matches(withText(weatherData.getWeather
```

```
State()));

onView(withId(R.id.weather_description)).check(matches(withText(weatherData.getW
eatherDescription())));

onView(withId(R.id.temperature)).check(matches(withText(weatherData.getTemperatu
reCelsius())));

onView(withId(R.id.humidity)).check(matches(withText(weatherData.getHumidity()))
);
    }
```

由于测试数据是预设的，不论网络是否通畅，都可以进行可靠的功能测试。执行测试，右键单击 MainActivityTest 文件，选择 Run 'MainActivityTest'，执行测试文件中的全部功能测试。当执行完成后，在 Android Studio 的底部，显示“All Tests Passed”、绿色进度条、“1 test passed”，表示功能测试全部执行成功，如图 8-4 所示。

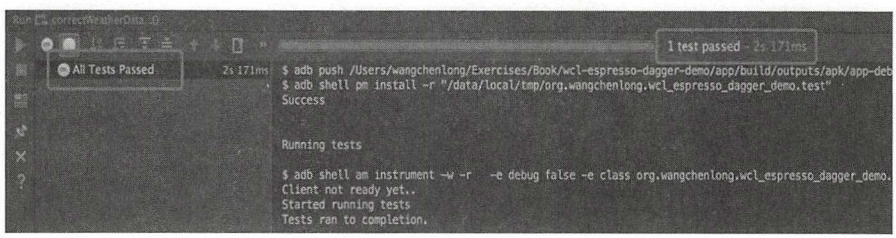


图 8-4　功能测试执行成功界面

在 Android Monitor 中，观察日志的 WCL-TEST 标记在执行 correctWeatherDataDisplayed() 方法之前执行 setUp()方法，这是测试的执行顺序。

```
E/WCL-TEST: setUp
E/WCL-TEST: correctWeatherDataDisplayed
```

至此，基于 Espresso 和 Dagger 的自动化测试框架已全部介绍完成。

在 Android 项目的开发过程中，如果是小型项目，不必或少量的编写测试用例即可，如果是多人参与的大型项目或超大型项目，则测试用例必不可少。Android 项目属于客户端项目，除了常见的业务逻辑测试以外，还需要界面逻辑测试。一般小型公司或者技术储备较弱的公司，可能还停留在通过测试人员进行页面测试，或者通过 Monkey 工具进行随机点击测试，无法有效地测试功能，还浪费大量人力。因此，引用自动化测试至关重要。本文所介绍的基于 Espresso 和 Dagger 的自动化测试框架，是目前主流的测试框架，在其之上，也可以扩展一些其他的测试组件，核心的测试方法万变不离其宗。通过 Dagger 注入不同的对象实例，可以方便地扩展逻辑，例如不同的网络数据源，分别用于线上和测试。通过 Espresso 模拟用户操作和验证显示数据，提供 perform()和 check()等测试接口。测试是软件开发至关重要的一步，高级软件开发人员的必

备技能之一。

8.2　优化内存泄漏与电量消耗的技术框架

对于 Android 应用而言，除了满足需求、实现功能以外，保证应用的质量和性能也是非常重要的。在应用的质量体系中，内存泄漏是 Android 开发中常见的问题，随着应用的长时间运行，泄漏的应用会不断增加内存的占用，直至临界点，导致应用突然崩溃，降低用户体验，使得程序的稳定性下降。在应用的性能体系中，电量消耗也是移动端应用不可避免的问题，因为手机的电量终究有限，操作系统会将应用的耗电量记录下来，高耗电应用可能会被一些定制的操作系统或者第三方管理软件警告，当电量消耗过快时，用户就有可能将高耗电应用卸载。

然而，在 Android 开发过程中，内存泄漏不可避免地会经常发生。当发生内存泄漏时，功能测试或单元测试无法测试，黑盒的测试人员也无从检查。虽然编程规范的普及非常重要，但是也需要一些检查内存泄漏的工具，帮助开发人员快速、准确地定位到内存泄漏的位置，及时修改，提升应用程序的稳定性。常用的内存泄漏检测工具有两个：Memory Analyzer Tool（MAT）和 Leak Canary。其中：

- ◎ Memory Analyzer Tool 是静态分析内存状态的工具，检测内存泄漏也是基于对于内存的分析结果。
- ◎ Leak Canary 是动态检测内存泄漏的工具，支持实时、快速地检测出内存泄漏的位置与影响，是非常实用的开发辅助工具。

同样地，对于应用的耗电量，也是用户的核心关注点，应用不仅要好用，还要尽可能地省电，因此，在一些高耗电的功能点上，就需要进行权衡，是功能点本身更重要，还是低耗电更重要。如果属于应用的核心需求，也可以设置高耗电功能开关，让用户自主选择，并提供低耗电功能的替代。常见的是，GPS 精准定位（高耗电）和网络模糊定位（低耗电）。

内存泄漏实例完整代码的下载地址为 https://github.com/SpikeKing/wcl-leakcanary-demo。

8.2.1　内存泄漏

关于检测内存泄漏的两个工具而言：Memory Analyzer Tool 更偏向于分析应用中静态的内存状态，而且操作较为复杂，检测内存泄漏只是其中的一个额外功能；Leak Canary 则是轻量级的辅助工具，专注于检测内存泄漏，支持集成于测试版本的应用中，快速、准确地定位于内存泄漏的位置。Memory Analyzer Tool 在早期是基于 Eclipse 开发环境的 Android 内存分析工具，现在已经独立成为一个工具模块。Leak Canary 是一个开源的第三方库，支持直接添加至 Android 工程中。关于 Memory Analyzer Tool 的使用方式，感兴趣的读者可以翻阅开发文档。由于本例

侧重于优化 Android 的内存泄漏问题，因此选择更具针对性的解决方案 ——Leak Canary。

在 Android 应用中，如果对于需要长时间保留的对象，其所持有的资源处理不当，就会导致内存泄漏，一般而言，主要的原因有两大类：

◎ 对于生命周期较长的类，如单例，如果持有 Activity 或 Service 的 Context，会导致 Activity 或 Service 长时间被引用，无法被及时回收和释放。当再次调用 Activity 或 Service 时，会创建新的实例，因此不断地占用资源，从而导致内存泄漏。当需要使用 Context 时，优先选择整个应用的 Context，即 Context.getApplicationContext()；如果一定需要持有 Activity 或 Service 的 Context，则需要在合适的时机合理地释放 Context 对象。

◎ 单例尽量不要持有页面的控件，因为这种情况下控件附属于页面，最终导致页面得不到释放，从而导致内存泄漏。如果单例需要修改控件属性，则在单例内部仅仅保留处理数据部分，修改逻辑仍位于页面中，通过访问数据修改控件属性。

因此，在 Android 开发中，一定要避免编写导致内存泄漏的代码，这是一些程序员经常会犯的开发错误之一。因此，需要在开发过程中及时发现和解决这种错误，这就需要用到检测内存泄漏的工具 Leak Canary。导致内存泄露的原因已经理解，接下来讲解一下如何在应用中检测内存泄漏。

1. 集成 Leak Canary

Leak Canary 源码地址为 https://github.com/square/leakcanary。Leak Canary 是以第三方库的形式集成于已有的工程中。在已有工程的 build.gradle 中，添加 Leak Canary 的依赖库。依赖库的类型含有两类：

◎ leakcanary-android 是在调试时编译，即 debugCompile。

◎ leakcanary-android-no-op 是在发布时编译，即 releaseCompile。

一般而言，仅需要在调试时编译，避免内存泄漏的产生，在正式发布时去除即可，提高应用的性能。

```
dependencies {
    debugCompile 'com.squareup.leakcanary:leakcanary-android:1.5.4'
    releaseCompile 'com.squareup.leakcanary:leakcanary-android-no-op:1.5.4'
}
```

注意：当工程的编译 SDK 版本（compileSdkVersion）低于 21 时，集成 Leak Canary 库会导致崩溃。错误提示：

```
Error:(2) Error retrieving parent for item: No resource found that matches the
given name 'android:Theme.Material'.
```

原因是在 Leak Canary 库中含有少量页面，是基于 Material Design 的设计样式，支持 SDK 的版本是 21（5.0）以上。因此，修改编译 SDK 版本（compileSdkVersion）为 21 及以上，即可解决。

在工程配置中添加 Leak Canary 依赖以后，还需在代码中启用 Leak Canary。创建 DemoApplication 继承于 Application，覆写入口函数 onCreate()，调用 LeakCanary 的 install()方法启用 Leak Canary 的检测。

```
public class DemoApplication extends Application {
    @Override public void onCreate() {
        super.onCreate();
        LeakCanary.install(this);
    }
}
```

同时，在工程清单 AndroidManifest.xml 的 application 标签中，将 name 属性设置为 DemoApplication，替换默认的 Application。当调用 DemoApplication 时，就会启动 Leak Canary 检测库。

```
<manifest>
    <application
        android:name=".DemoApplication">
    </application>
</manifest>
```

如果关闭 Leak Canary 也非常简单，在 DemoApplication 的 onCreate()中，移除 Leak Canary 的启动代码即可。当集成 Leak Canary 之后，再次启动应用时，就会额外创建一个 Leak 应用，图标是黄底黑影，用于显示内存泄漏的信息，如图 8-5 所示。

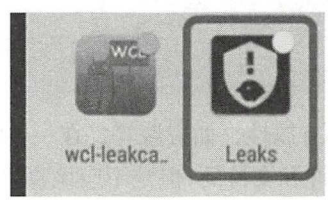


图 8-5　显示内存泄漏的 leak 应用

2. 错误单例

在 Android 开发中，单例是最容易导致内存泄漏的地方，变量持有不当就会导致无法释放，从而与变量关联的多个资源都被持有，造成内存泄漏。本节创建一个经典的泄漏单例 LeakSingle，在单例中，含有两处内存泄漏的点：

◎　直接持有 Context，没有将 Context 转换为 Application 的 Context，再持有，导致

Activity 或 Service 实例无法释放，产生内存泄漏。

◎ 引用视图控件 TextView，就会强制保留视图控件的 Activity 实例，产生内存泄漏。

这个 LeakSingle 单例设计混乱，非常容易导致内存泄漏。

```
public class LeakSingle {
    private static LeakSingle sInstance;  // 单例

    private Context mContext;  // 单例持有 Context
    private TextView mTextView;  // 单例持有视图控件

    private LeakSingle(Context context) {
        mContext = context;  // 容易导致内存泄漏
//        mContext = context.getApplicationContext();  // 正确写法
    }

    public static LeakSingle getInstance(Context context) {
        if (sInstance == null) {
            sInstance = new LeakSingle(context);
        }
        return sInstance;
    }

    // 持有视图控件，容易产生内存泄露
    public void setRetainedTextView(TextView tv) {
        mTextView = tv;
        mTextView.setText(mContext.getString(R.string.app_name));
    }

    // 删除引用，防止泄露
    public void removeRetainedTextView() {
        mTextView = null;
    }
}
```

在 Context 中，含有应用的特有属性，如资源等，在一些情况下，必须要持有 Context，不可避免。因此，持有 Context 是被允许和支持的，但是不应该直接持有 Activity 或 Service 的 Context，导致实例无法被释放，不断重复创建，产生内存泄漏。正确的做法是，持有 Application 的 Context，因为 Application 实例是伴随着应用的生命周期而存在的，与单例的存在时间相同，不会导致内存泄漏。

持有 Context 的正确写法是：

```
private LeakSingle(Context context) {
//    mContext = context;  // 容易导致内存泄漏
    mContext = context.getApplicationContext();  // 正确写法
}
```

在视图控件中，如 TextView，持有页面 Activity 的 Context，当单例持有视图控件时，就是间接持有 Activity 的 Context，同样会导致实例无法被释放，产生内存泄漏。一般而言，单例只应该处理事务性的工作，控件操作应该使用接口方法进行处理，尽可能地避免直接引用视图控件，即持有视图控件不被允许和支持。但是，如果在精力有限或者业务理解较低时，可以选择替代方案，即在 Activity 的 onDestroy()时，解除视图控件的引用，避免长时间持有。

持有视图控件，如 TextView，的替代写法是：

```
// 持有视图控件，容易产生内存泄露
public void setRetainedTextView(TextView tv) {
    mTextView = tv;
    mTextView.setText(mContext.getString(R.string.app_name));
}

// 删除引用，防止泄露
public void removeRetainedTextView() {
    mTextView = null;
}
```

本示例仅仅作为反面示例，用于测试 Leak Canary 的内存泄漏检测功能，切勿模仿学习。

3. 内存泄露

在 MainActivity 的入口函数 onCreate()中，创建单例 LeakSingle。在单例 LeakSingle 中：

◎ 调用 getInstance(this)，导致 Context 类型的内存泄漏，this 表示当前的 Activity，即在单例中，直接绑定 Activity 的 Context，导致 Activity 无法释放。当每次启动 Activity 时，都会重复创建页面，产生内存泄漏。

◎ 调用 setRetainedTextView(mTvText)，在单例中持有 TextView 视图控件 mTvText，设置 TextView 的文本内容为 R.string.app_name，这样就会导致单例持有视图控件，间接地持有 Activity 的 Context，导致 Activity 无法释放，产生内存泄漏。

这样，在创建 LeakSingle 时，就会导致两种情况的内存泄漏。

```
public class MainActivity extends AppCompatActivity {
    @Bind(R.id.main_tv_text) TextView mTvText;  // 首页的文本控件

    @Override protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ButterKnife.bind(this);

        // 1. Context 导致泄漏，2. 视图控件导致泄漏
        LeakSingle.getInstance(this).setRetainedTextView(mTvText);
    }
}
```

接着，启动应用，打开 MainActivity 页面，再关闭 MainActivity 页面，由于单例始终持有 MainActivity 页面，导致页面的资源无法被释放，造成内存泄漏。由于工程已经集成 Leak Canary 插件，就会额外生成一个显示内存泄漏的辅助应用 Leak。当发现应用存在内存泄漏时，以应用通知的形式告知开发者，当前操作导致内存泄漏，在通知中显示泄漏的页面和占用的内存，如“MainActivity Leaked 1.2 kB”，如图 8-6 所示。

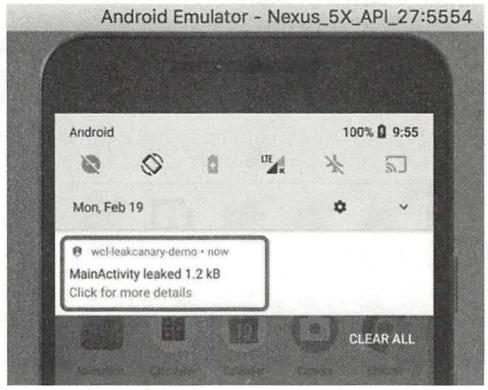


图 8-6　在通知中显示泄漏的页面和占用的内存

点开通知，获得关于此次内存泄漏的详情。通过详情中的信息，可以得知此次内存泄漏的位置：

◎　以线程的 Context 类加载器为起点，即 Thread.contextClassLoader。
◎　引用运行时内部对象，即 runtimeInternalObjects。
◎　引用对象数组，即 Object[]。
◎　引用单例静态变量 sInstance，即 LeakSingle.sInstance。
◎　引用 Context 变量 mContext，即 LeakSingle.mContext。
◎　最终，导致 MainActivity 的实例（instance）泄漏。

静态变量一般都是导致内存泄漏的罪魁祸首，在使用时一定要谨慎。Java 语言虽然一般不用考虑类的回收问题，但是对于静态变量，却无法控制其生命周期。通过 Leak Canary 内存泄漏的提示，在代码中结合已有的编程经验，发现内存泄漏的原因，即单例引用 Activity 的 Context，修改代码以解决这个问题，如图 8-7 所示。

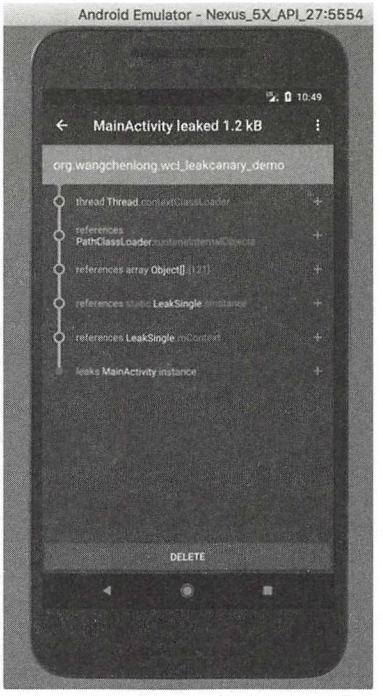


图 8-7　内存泄漏原因（一）

单例引用的 Context，应该选择 Application 的 Context，要么在调用时选择，即 this.getApplication()，要么在持有时选择，即 context.getApplicationContext()。如果要想彻底地避免 Context 的内存泄漏，在调用和持有时都要选择 Application 的 Context，避免可能导致内存泄漏的情况。例如，将创建单例的方式修改为 getInstance(this.getApplication())，即使用 Application 的 Context 作为参数传递至单例，这样就解决 Context 类型的内存泄漏问题。根据经验可知，单例还是存在非合理引用视图控件的问题，仍然会导致内存泄漏。

```
LeakSingle.getInstance(this.getApplication()).setRetainedTextView(mTvText);
```

同样地，启动应用，打开 MainActivity 页面，再关闭 MainActivity 页面，仍然存在内存泄漏的问题。点开通知，获得关于第二次内存泄漏的详情，定位内存泄漏的位置：

◎　前三步与上一次内存泄漏类似，都是以 Context 为起点。
◎　引用静态变量，即 LeakSingle.sInstance。
◎　引用单例静态变量 sInstance，即 LeakSingle.sInstance。
◎　引用视图控件 mTextView，即 LeakSingle.mTextView。
◎　引用 Context 变量 mContext，即 AppCompactTextView.mContext。
◎　引用 Context 变量 mBase，即 TintContextWrapper.mBase。

◎ 最终导致 MainActivity 的实例（instance）泄漏。

静态变量仍然是导致内存泄漏的罪魁祸首，核心还是 Context 的引用问题，源于 TextView 控件实例是也是基于 Activity 的 Context，当单例持有 TextView 实例时，间接持有 MainActivity 实例产生内存泄漏，如图 8-8 所示。

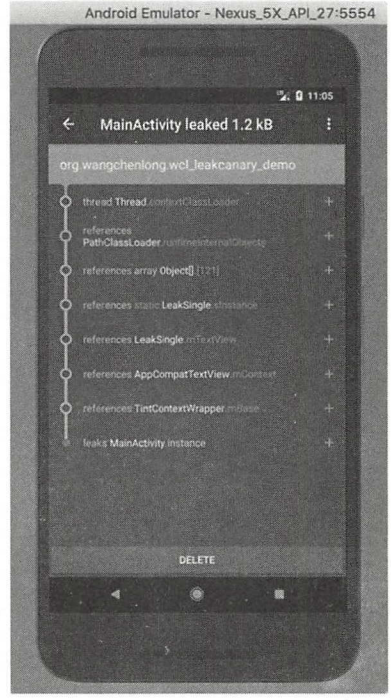


图 8-8 内存泄漏的原因（二）

最优的解决方案是在单例中只执行与数据相关的操作，不执行与具体页面相关的操作。替代的解决方案是在单例中添加释放视图控件资源的接口 removeRetainedTextView()，在 Activity 的 onDestroy()中调用接口，释放持有的视图控件资源。这样就可以避免内存泄漏的产生。

```
public class LeakSingle {
    // 删除引用，防止泄露
    public void removeRetainedTextView() {
        mTextView = null;
    }
}

public class MainActivity extends AppCompatActivity {
    @Override protected void onDestroy() {
        // 防止内存泄露
```

```
LeakSingle.getInstance(this.getApplication()).removeRetainedTextView();
            super.onDestroy();
        }
    }
```

当解决全部问题以后，启动应用，打开 MainActivity 页面，再关闭 MainActivity 页面，则 Leak 应用不会报警和给予消息提示，证明两个内存泄漏的问题都完美地解决了。

当应用比较复杂、页面较多时，可能从起始页面到待验证页面，需要额外启动很多页面，层级较多，验证起来也比较麻烦。这时就可以使用 ADB 的 shell 命令 “am start –n”，直接启动待验证的页面。

```
adb shell am start -n [包名]/[Activity 名]
```

例如，在本例中，启动 MainActivity 的命令，如下：

```
➜  ~ adb shell am start -n org.wangchenlong.wcl_leakcanary_demo/org.
wangchenlong.wcl_leakcanary_demo.MainActivity
Starting: Intent { cmp=org.wangchenlong.wcl_leakcanary_demo/.MainActivity }
```

当第一次关注内存泄漏时，如果需要排查应用中全部内存泄漏的问题，一般而言，需要启动和关闭应用中全部 Activity，通过 Leak Canary 检查页面是否发生内存泄漏。排查完毕之后，在调试编译时，集成 Leak Canary 插件，随时发现内存泄漏，及时修复即可。

4. 其他信息

在 Leak 应用的内存泄漏页面中，点击左上角的后退按钮，回退至上一页，就到达内存泄漏的列表页。每一项就是每一次内存泄漏的具体信息，含有泄漏的页面、泄漏的内存、泄漏的时间等。列表是按时间顺序排列的内存泄漏记录，如图 8-9 所示。

在内存泄漏页面中，点击右上角的菜单按钮，还支持将内存泄漏信息分享出去，用于统计和记录。“Share info” 是分享内存泄漏的信息；“Share heap dump” 是分享内存泄漏的堆栈信息，如图 8-10 所示。

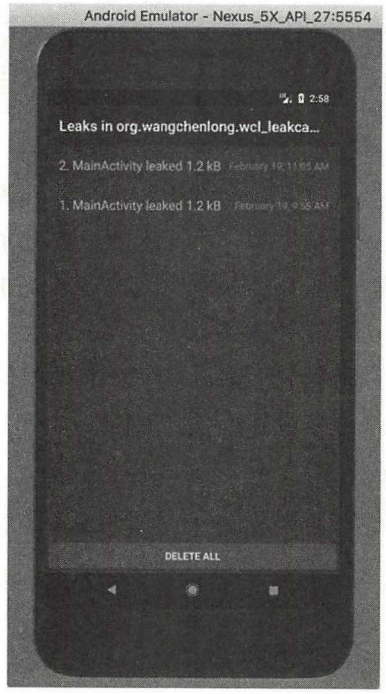


图 8-9　内存泄漏的列表页

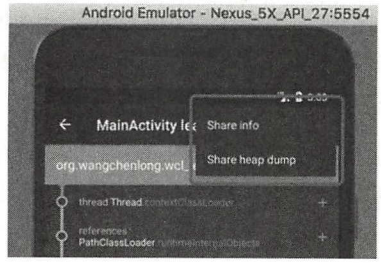


图 8-10　分享内存泄漏信息

在内存泄漏页面中，点击每一项的加号“+”按钮，可将简写的路径类变为含有包名的路径类，方便更加准确地查找内存泄漏的位置，如图 8-11 所示。

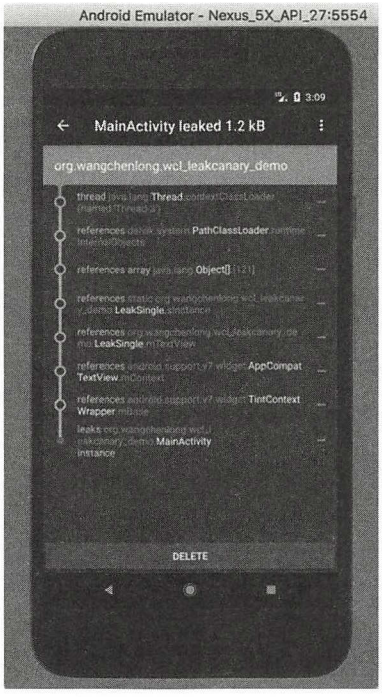


图 8-11　将简写的路径类变为含有包名的路径类

在内存泄漏页面中，点击底部的删除“DELETE”按钮，就可以将内存泄漏记录删除。

解决内存泄露的问题对于应用的用户体验至关重要，感谢 Leak Canary 产品，功能强大，使用简单，让处理这件事情变得如此简单。

8.2.2　电量优化

在 Android 项目中，较难监控应用的电量消耗，然而用户却非常关心手机的待机时间，希望手机的使用时间尽可能的长。过度耗电的应用会遭到用户无情地卸载，对电量要重视，不要存在侥幸的心理，以防给竞品带来机会。因此，需要分析应用的耗电原因，在不影响核心功能的情况下，适度地优化和降低应用的耗电量。

本文主要介绍的工具是 Battery Historian，它是一款由 Google 公司提供的 Android 系统电量分析工具。通过输入手机的电量详情文件，Battery Historian 支持在可视的网页中展示手机电量消耗的过程。关于 Battery Historian 的更多信息，可以参考：**Battery** Historian：https://github.com/google/battery-historian。

只有掌握应用的耗电情况，才能更好地优化应用的电量，也可以把耗电量作为应用的性能

测试指标。

1. 安装 Go 语言

Battery Historian 是基于 Go 语言开发的，因此运行 Battery Historian 需要提前安装 Go 语言编译环境。Go 语言是一种支持并发的、带有垃圾回收的、可快速编译的编程语言。同时，Go 语言也是一种编译型语言，结合解释型语言的灵活善变、动态类型语言的开发效率、静态类型的、安全性的编程语言。

```
Go: http://golang.org/
```

首先，下载 Go 语言的 Mac 版本安装包，点击即可执行。

Go 语言安装包的下载地址为 http://golang.org/doc/install。

Go 语言安装完成的页面如图 8-12 所示。

图 8-12　Go 语言安装完成的页面

当 Go 语言安装完成后，执行“go version”，检查当前系统的 Go 语言版本。

```
➜  ~ go version
go version go1.10 darwin/amd64
```

同时，在系统配置.bash_profile 中，设置 Go 语言的环境变量，以便直接调用 bin 中的命令。

```
#Go Settings
export GOPATH=/Users/.../Workspace/GoWorkspace
export GOBIN=/Users/.../Workspace/GoWorkspace/bin
```

GOPATH 是 Go 语言的源码地址，GOBIN 是 Go 语言的可执行文件地址。GOBIN 路径也可以设置为$GOPATH/bin，与 GOPATH 路径统一。接着，执行 source .bash_profile，启动 bash_profile 中的系统配置。

在 GOPATH 路径下，即 GoWorkspace 文件夹中创建 src 文件夹，添加 Go 语言的 HelloWorld 文件 hello.go，用于验证 Go 语言的编译环境是否可用。在 hello.go 中，添加如下代码：

```
package main
import "fmt"
func main() {
    fmt.Printf("hello, world\n")
}
```

通过命令“go install”安装 hello.go，即在 GOBIN 中，生成 hello 可执行文件。

```
go install hello.go
```

最后，执行 hello 可执行文件，即

```
$GOBIN/hello
```

如果在终端中显示“hello, world”，即表示 Go 语言的编译环境已安装完成。Go 语言是非常优秀的开发语言，也是值得掌握的开发语言。完成本例仅需要提供 Go 语言支持即可。

2. 运行 BH

在安装 Battery Historian 工具之前，需要提前安装 wget。wget 是一个自动下载文件的工具，支持通过 HTTP、HTTPS、FTP 三个 TCP/IP 协议下载，同时支持 HTTP 代理。如果不支持 brew 工具，也需要提前下载 brew。下载 wget 的命令如下：

```
sudo brew install wget
```

接着安装和运行 Battery Historian 工具，逐步执行以下安装和启动命令即可。

```
go get -d -u github.com/google/battery-historian/...
cd $GOPATH/src/github.com/google/battery-historian
go run setup.go
go run cmd/battery-historian/battery-historian.go [--port <default:9999>]
```

需要注意的是，在执行配置文件 setup.go 时耗时较长，因此 Closure 库较大，下载速度可能较慢，需要耐心等待。如果下载失败或下载异常，可以删除“third_party”文件夹，重新执行配置文件 setup.go。

```
➜  battery-historian git:(master) go run setup.go

Downloading Closure library...

Downloading Closure compiler...

Downloading 3rd-party JS files...

Generating JS runfiles...
```

```
Generating optimized JS runfiles...
```

当 Battery Historian 工具安装完成后，每次只需执行以下启动命令，即可启动 Battery Historian。

```
cd $GOPATH/src/github.com/google/battery-historian
go run cmd/battery-historian/battery-historian.go [--port <default:9999>]
```

命令的演示操作如图 8-13 所示，Battery Historian 工具在本地的 9999 端口下运行。

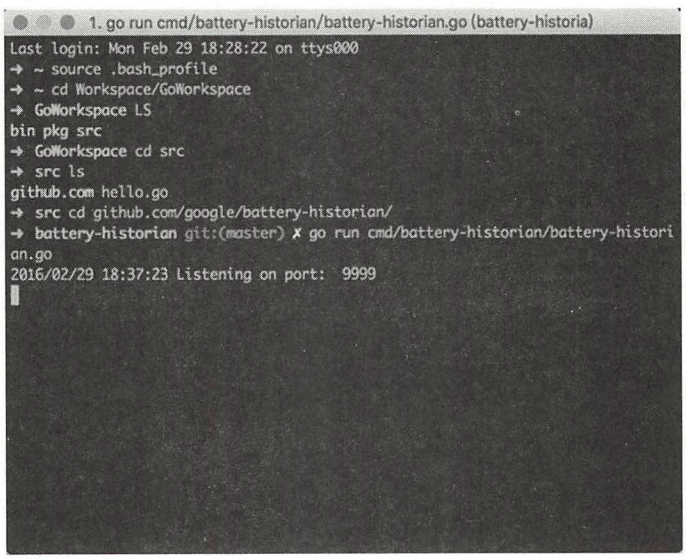


图 8-13 命令的演示操作

当启动完成后，在浏览器中输入 http://localhost:9999/，即可启动电量检测页面，如图 8-14 所示。注意，在启动页面时，可能需要连接 VPN，用于访问 Google 公司提供的一些配置信息。浏览器也需要适配，如果 Chrome 浏览器无法使用，可以尝试切换 Safari 浏览器。

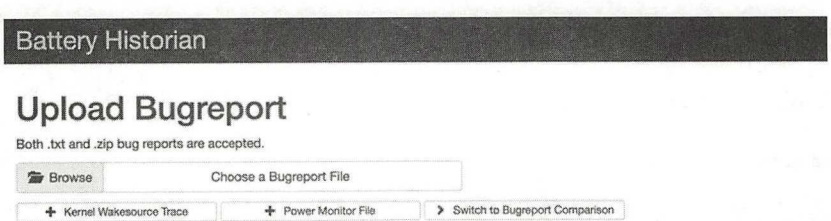


图 8-14 电量检测页面

除此之外，也可以选择一些其他开发者部署的在线网址。属于第三方开发者维护的网址，持续性和一致性不一定有保证。如

https://bathist.ef.lc/。

Battery Historian 工具用于分析手机电量的消耗情况，因此需要将手机耗电信息的文件下载至本地，并上传至本地 Battery Historian 的 Web 网址中。

Android 操作系统都含有电量监控的功能，一般位于【Settings】/【Battery】位置。其中，含有 Battery usage 的选项，即显示电量消耗信息。模拟器默认处于充电状态，无法显示电量消耗。在电量页面中，除了显示电量消耗信息，一般还提供设置省电模式、显示剩余电量百分比、设置睡眠时间等功能，如图 8-15 所示。

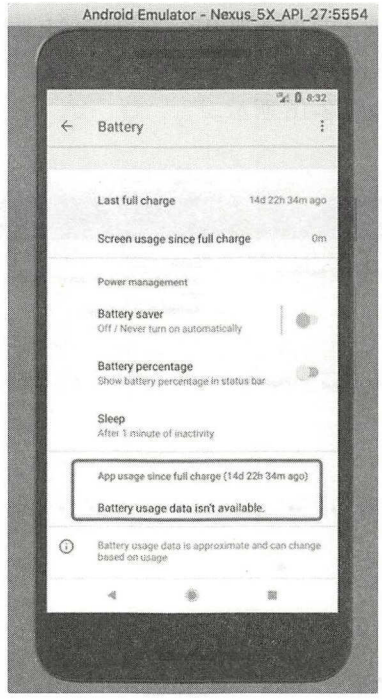


图 8-15　电量监控功能界面

3. 分析电量

首先，使用 adb 的 bugreport 命令获取手机状态的文件 bugreport.txt，同时将文件导出到根目录，以备 Battery Historian 工具使用。

```
adb bugreport > bugreport.txt
```

如果无法执行 bugreport 命令，或出现如下异常，可能的原因就是 adb 的版本不兼容，即高版本 adb 无法获取低版本手机中的 bugreport 信息。

```
    Failed to get bugreportz version, which is only available on devices running
Android 7.0 or later.
    Trying a plain-text bug report instead.
```

下载低版本的 platform-tools 即可，注意下载源中的操作系统类型，Mac 的后缀是 macosx、Windows 是 windows、Linux 是 linux。

```
    https://dl.google.com/android/repository/platform-tools_r23.0.1-macosx.zip
```

下载完成后解压文件，替换 android-sdk 中相应的 platform-tools 文件夹即可。

获取手机中的 bugreport 信息，可能需要一段时间，请耐心等待。

接着，使用 Battery Historian 工具的本地 Web 网页加载 bugreport.txt 文件。如遇到一些问题导致加载失败，则重新提交 bugreport.txt 文件即可。当加载成功之后，就可以检查系统和应用的耗电状态。电量分析的效果如图 8-16 所示。

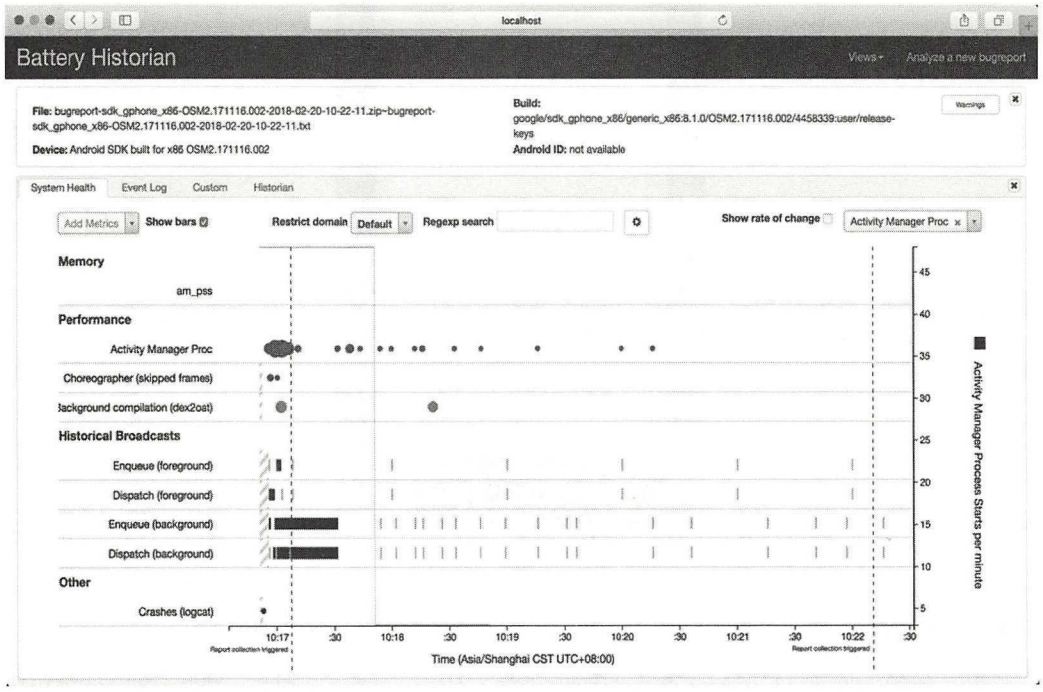


图 8-16　电量分析的效果

根据 Battery Historian 的电量状态信息，高耗电的项目包含唤醒锁、同步管理器（SyncManager）、音视频、流量等。根据多年的开发经验，提供一些电量的优化方式，仅供参考：

（1）优化唤醒锁（Wakelock）。检查全部唤醒锁，是否存在冗余或者无用的唤醒锁，唤醒锁会导致手机的 CPU 无法休眠，长时间工作是浪费电量的关键所在。

（2）优化网络请求。集中具有相关性的网络数据请求，统一发送；精简数据结构，减少无用数据的传输，这样不仅可以节省电量，也能节省流量。

（3）优化数据处理。对于分析和统计等非重要的计算，在电量充足或连接 WIFI 时进行，可以通过 JobScheduler 类进行处理。

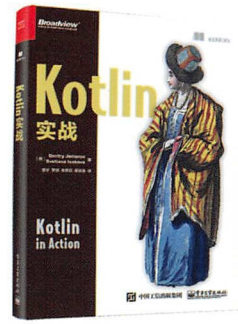
（4）优化服务。精简冗余的服务（Service），避免长时间执行耗电较高操作。

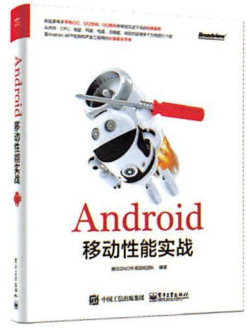
（5）优化定位。注意精准定位（GPS）信息的获取，使用后及时关闭，对于精度要求较低的定位，可以通过网络定位进行，以便降低耗电量。

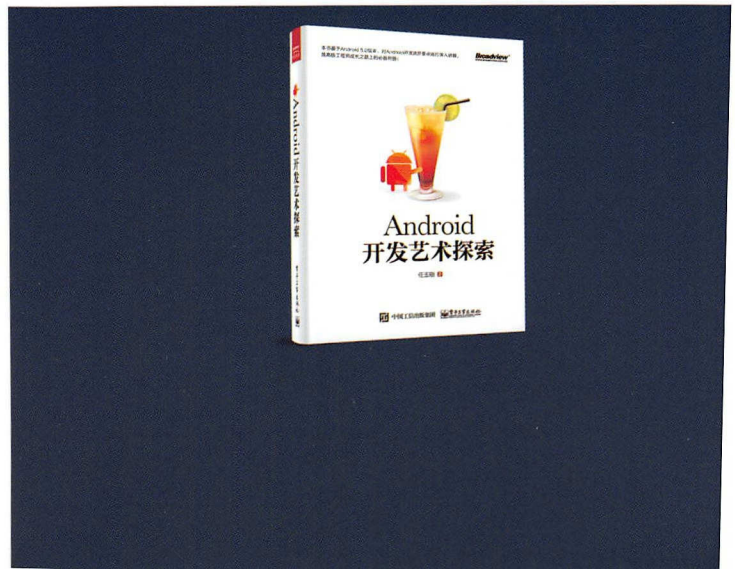
电量优化并不是很难，但需要对业务非常熟悉，应了解一些耗电操作的使用情况，并且及时优化。只有给予精致的用户体验，才能让用户更加喜欢我们的应用，这就是产品服务的本质。

本节讲解了应用性能优化的两个最关键的点，即避免内存泄漏和合理降低电量。不仅提供了检查问题的工具，即 Leak Canary 和 Battery Historian，还提供了具体问题的解决方案。内存泄漏的危害是无端导致应用内存占用过大，导致应用崩溃；高耗电量的危害是无意义地浪费系统电量，导致移动终端电量不足，导致用户对产品失去信心。在具体的开发中，内存泄漏一定要尽量避免，如果可能，应保持零泄漏，但是电量消耗需要针对于产品进行优化。例如，对于一款计步应用，如果要保持计步准确，就一定需要精确地定位 GPS，低端设备需要长时间唤醒传感器，这样耗电量就会大幅增加，但是如果不这样做，就会导致计步效果较差。在这种情况下建议将权利交给用户，如果用户关注于产品效果，则选择开启，如果用户在意手机电量，则选择关闭。不要替用户强制选择，给予用户的空间。

真正的高级 Android 工程师，需要做的不仅仅是完成产品功能，要更多从产品性能和产品功能之间做出权衡。将产品性能优化到极致，将产品功能开发到完美，有时可能在两者之间选择。不仅负责开发应用的功能，还要尽可能地从架构设计入手，完善整个应用。合理地引入开源工具，帮助完善产品。对于产品的极致追求，是工程师的最终信仰。







Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

高级Android开发强化实战

与市面上的入门书不同，本书侧重于项目实战，并引入了实战中较前沿的知识，如Kotlin、项目架构、自动化测试框架等。相信这些知识能帮助更多的中级工程师向高级工程师迈进，值得一看！

徐烨　美团高级Android开发工程师

本书通过大量的实例，将晦涩抽象的知识点更清晰、直观地进行展现，相信读者会有茅塞顿开和“so easy”之感。本书对于架构和性能优化也有清晰的介绍和归纳，利于读者提高技术水平，很适合期待进阶为高级或资深Android工程师的有识之士阅读。

王泽文　快手资深Android开发工程师

本书深入浅出，从开发模式到常用的流行框架和性能优化方法，都进行了由表及里的深入分析，对于Android进阶非常有益。本书结合实践案例进行讲解，摒弃了枯燥的原理陈述，可让读者在具体场景下了解Android技术。

张云华　网易云音乐资深Android开发工程师

本书是作者对Android格物致知的结晶。全书采用专题形式进行讲解，非常方便读者按需所取、专项突破。本书在内容选取上既有对基础知识深入浅出的原理分析，也有对工程架构的实践探讨，更有对Android热门、前沿知识的讲解。希望读者不仅可以收获一个个专题，更能体会到作者既有深度又有广度，并在此基础上构建技术体系的学习方法。

吴林　春雨医生资深Android开发工程师

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview
策划编辑：张国霞
责任编辑：宋亚东
封面设计：侯士卿
上架建议：移动开发>Android
ISBN 978-7-121-34298-1
9 787121 342981 >
定价：89.00元